平成 23 年 11 月 20 日
京都工芸繊維大学コンピュータ部

Lime44

はじめに

こんにちは、部長の原です。今年も無事 Lime を発行でき嬉しく思います。さて、Lime も今回で第 44 回です。4 が 2 つ続いています。Lime は、部員の活動の一部を記したものです。日々の活動の成果や興味に対する調べ物、個人的趣味全開なものを記しています。

是非、最後までお付き合いください。ではでは

平成 23 年 11 月 20 日コンピュータ部部長　原　一貴

目 次

1 はじめてのSchemeコンパイラ

京都大学大学院 情報学研究科 修士 2 回 湯浅 信吾

1.1 はじめに

今からちょうど 5 年前。当時、学部 1 回生だった私は murasaki[1] という Scheme インタプリタを C 言語で書きました。これが「はじめての Scheme インタプリタ」でした。その後、JavaScript、NScripter、PHP、PostScript、Prolog、ニコスクリプトなどの言語で様々な Lisp/Scheme インタプリタを書いてきました。私の大学生活は Lisp/Scheme インタプリタと共にあったといっても過言ではないかもしれません。しかし、思い返してみれば、コンパイラを書いたことは一度もなかったのです。6 年もの大学生活で私は一体何をやってきたのでしょうか。今からでも遅くない。学生最後の夏休みも後半に入った 9 月 13 日。私は「はじめての Scheme コンパイラ」を書き始めました。

1.2 Scheme の事情

まずは、Scheme をあまり知らない人のために、Scheme とはどんなプログラミング言語なのか説明したいと思います。ほんの 50 ページもあれば、Scheme のすべてを説明できる [2] のですが、紙面の都合上、それは叶いません。そこで、本稿では他の言語にはあまり見られない、Scheme ならではの特徴のみを説明することにします。

1.2.1 Properly Tail-recursive Implementation

Scheme では、手続き[1]の最後で行う手続き呼び出しを **tail call** と呼びます。言葉で説明するよりコードを見た方が分かりやすいでしょう。図 1.1 を御覧ください。手続き `proc1` はまず手続き `display` を呼び出して、画面に “hello” と表示します。`proc1` はまだやることが残っているので、この `display` の呼び出しは tail call ではありません。`proc1` は次に、手続き `proc2` を呼び出します。この `proc2` の呼び出しは `proc1` の最後で行なっているので tail call です。手続き `proc2` は構文 `if` を用いて `x` が 0 の場合と、それ以外の場合で処理を分岐しています。`x` が 0 の場合は “world” を表示して、それ以外の場合は “work” を表示します。この 2 つの `display` の呼び出しは両方とも tail call です。1 つめの `display` の呼び出しはソースコードの字面的には手続きの最後とは言い切れませんが、`if` の意味を考えると、「手続き `proc2` の最後の仕事」であるため tail call となるのです。このように、`if` などの構文が出てくると、tail call の定義は少し分かりにくくなりますが、それでも直感的に「これで最後だ！」と分かると思いますので、ここではこれ以上厳密な定義はしないことにします。

tail call を行った先の手続きでまた tail call をして、その先の手続でまた tail call をして、といった具合に、tail call を無限に行える処理系を **properly tail-recursive** であると言います。Scheme 処理系は properly tail-recursive でなければなりません。これも言葉ではわかりにくいのでコードを使って説明します。図 1.2 を御覧ください。手続き `loop` は引数 `n` が 0 以下の場合は画面に “return from loop” と表示して、それ以外の場合は `n` から 1 を引いたものを引数として自分自身（`loop`）を呼び出します。引数 `n` は自分自身を呼び出すたび

---

[1] 他の言語の「関数」と同じものと思ってください。本稿では「手続き」というと Scheme の手続きを意味して、「関数」というと C 言語の関数を意味することにします。

```
(define (proc1 x)          ; 引数 x をとる手続き proc1 を定義
  (display "hello")        ; 画面に hello と表示
  (proc2 x))               ; x を引数として手続き proc2 を呼び出す

(define (proc2 x)          ; 手続き proc2 を定義
  (if (= x 0)              ; x が 0 かどうかで条件分岐
      (display "world")    ; x が 0 なら world と表示
      (display "work")))   ; それ以外なら work と表示
```

図 1.1: Scheme の簡単なプログラム

```
(define (loop n)                     ; 手続き loop を定義
  (if (<= n 0)                       ; 引数 n の値で条件分岐
      (display "return from loop")   ; n が 0 以下なら画面に出力
      (loop (- n 1))))               ; それ以外の場合は再度 loop を呼ぶ
```

図 1.2: tail call を繰り返すプログラム

に 1 ずつ減少していくため、いつかは 0 になり、プログラムが停止します。この `loop` の呼び出しは tail call です。properly tail-recursive な処理系は、tail call を無限に出来なければならないので、どれだけ大きな `n` を与えても、手続き `loop` は必ず停止します。

手続き `loop` が必ず停止するというのは当たり前のことに思えますが、多くのプログラミング言語は、その処理系が properly tail-recursive であることは要求しません。例えば、C 言語や Java で上記 `loop` のような関数・メソッドを定義した場合、`n` が非常に大きい場合はスタックがあふれて正しく動作しなくても[2]何の問題もないのです。実際にためしてみると恐らく上手くいかないでしょう。しかし、Scheme では必ず正しく動作します。

1.2.2 First-class Continuation

「これからやろうとしている処理」のことを **continuation** と呼びます。例えば、$1 + 2 \times 3$ を計算するプログラムを考えてみます。このプログラムはまず 2×3 を計算してから、それに 1 を加えます。ここで、2×3 の計算が終わった瞬間を考えてみてください。この瞬間においての continuation、つまり「これからやろうとしている処理」とは「計算結果に 1 を加える」です。図 1.3 を御覧ください。`call/cc` という手続きを用いて、continuation を取得して、それを変数 `cont` に代入した後に、2×3 を計算しています。これを実行すると $1+2\times 3$ の値である 7 が表示されます。変数 `cont` には「計算結果に 1 を加えたあと表示する」という continuation が入っているため、このあとに `(cont 99)` としてやると、「計算結果は 99 だった」と記憶が改ざんされた上で計算が再開され、その結果、画面に 100 が表示されます。

このように、Scheme は continuation を変数に代入したり、手続きに渡したり、自由に扱えるので、Scheme の continuation を first-class continuation と呼びます。これをうまく使うことで、今やってる仕事を放棄したり（大域脱出)、今の仕事を保留して、別の仕事をやってから、保留していた仕事を再開するようなプログラムが実現できます。つまり、マルチスレッドのような処理が、ユーザプログラムとして書けるという訳です。大域脱出は多くのプログラミング言語で実現されていますが、処理の再開をできるような言語は Scheme 以外にはあまりありません。

```
(define cont #f)                ; 大域変数 cont を作る
(display
 (+ 1 (call/cc (lambda (k)      ; continuation を k として受け取る
                 (set! cont k)  ; k を cont に代入
                 (* 2 3)))))    ; 2*3 の値を求める
```

図 1.3: continuation を capture するプログラム

[2]正しく動作しないというのは曖昧な表現ですが「ここでは “return from loop” を表示して停止する」くらいの意味として捉えてください。

1.2.3 Scheme から C へのトランスレータを考える

処理系が properly tail-recursive であることや、first-class continuation を扱えることは、非常に便利であるため、Scheme を使ってプログラムを書く人には嬉しいばかりです。しかし、Scheme の処理系を書く人には悩みの種でもあります[3]。

コンパイラを作る際に、ターゲットのプログラムから、直接アセンブリ言語や機械語のプログラム生成するのではなく、C 言語のプログラムを生成するというやり方があります。例えば `(define (f x) (+ x 1))` という Scheme のプログラムから、`int f(int x) {return x + 1;}` という C のプログラムを生成するのです。この手法を用いると、コンパイラの移植性が非常に高くなりますし、C 言語のコンパイラの最適化が優秀であれば、速度の面でも優れた性能を発揮できます。しかし、C 言語には first-class continuation なんてありませんし、C 言語のコンパイラが properly tail-recursive になっているなど考えられません[4]。Scheme のプログラムを C のプログラムに変換するだなんて、トンビにタカを生めと言うようなものです。

1.3 Scheme コンパイラの作り方

それでは Scheme のプログラムを C のプログラムに変換することは出来ないのでしょうか。もちろん、そんなことはありません。CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. [3] という変わったタイトルの論文がありまして、Scheme のプログラムを C に変換する**非常に面白い**手法が書いてあります。本節ではこの論文の手法について説明します。

1.3.1 CPS 変換

まず、論文の中身に入る前に、前提知識である **CPS 変換**について説明します。CPS とは「次にやること（continuation）を引数として渡す（passing）やり方 (style)」のことです。CPS 変換とは、読んで字の如し、普通のプログラムを CPS に変換することです。手続き/関数というものは基本的に、呼び出したら（call）、いつかは戻ってくる（return）するものですが、CPS はその基本的な考えを覆します。

最初に、continuation は手続きとして表現するようにします。例えば、少し前に出てた「計算結果に 1 を加える」という continuation は、`(lambda (x) (+ 1 x))` となるわけです。継続を起動してやるためには手続き呼び出しをすればいいわけです。「2×3 の結果」を上記継続に渡す場合は `((lambda (x) (+ 1 x)) (* 2 3))` となります。そして、手続きは本来の引数に加えて continuation を受け取るように変換します。例えば、`(define (f x) x)` は `(define (f cont x) (cont x))` となります。値を返すのではなく、値を継続に渡してやるのです。CPS 変換に関しては、詳しい説明が去年の Lime[4] にありますので、ここでは変換規則の一覧だけを図 1.4 に載せておきます。この 21 世紀にあえて M 式風の記法で載せていますが深い意味はありません。式 `exp` を CPS 変換した結果は `CPS-TRANS[exp]` の値となります。

1.3.2 Cheney on the M.T.A.

Scheme のプログラムに CPS 変換を施すと、任意の時点での continuation を手続きとして自由に扱えるようになります。これで、first-class continuation は実現できます。しかし、これだけではまだ問題があります。C 言語のコンパイラは properly tail-recursive ではないので、関数呼び出しを繰り返し、戻らない（return を使用しない）という方針をとると、いつかはスタックがあふれてしまいます。ここで Cheney on the M.T.A. の出番です。この手法の基本的な考え方は簡単です。まず、素直に CPS 変換した関数を使うことにより、スタック

[3]私が 5 年前に Scheme インタプリタを作った際には、properly tail-recursive への対応を後回しにしたために、大部分のコードを書き直すはめになってしまいました。
[4]GCC は tail call を jmp に変換する最適化を行うことができます。しかし、これも完璧ではなく、場合によっては最適化が行われません。C 言語はポインタを使って変数のエイリアスを作れてしまうため、安易には tail call を jmp に変換することはできないのです。

```
CPS-TRANS[exp] = CPS[exp; (lambda (x) x)]
CPS[atom; cont] = (cont atom)
CPS[(if exp1 exp2 exp3); cont] =
  CPS[exp1; (lambda (y) (if y CPS[exp2; cont] CPS[exp3; cont]))]
CPS[(quote exp); cont] = (cont (quote exp))
CPS[(set! var exp); cont] =
  CPS[exp; (lambda (z) (cont (set! var z)))]
CPS[(lambda args body); cont] =
  (cont (lambda (c . args) CPS[body; c]))
CPS[(op arg1 arg2 ...); cont] =
  CPS-EVLIS[REV[(op arg1 arg2 ...)]; REV[(f x y ...)]; (f cont x y ...)]
CPS-EVLIS[(); (); acc] = acc
CPS-EVLIS[(a . args); (p . params); acc] =
  CPS-EVLIS[args; params; CPS[a; (lambda (p) acc)]]
REV[xs] = REVI[xs; ()]
REVI[(); acc] = acc
REVI[(x . xs); acc] = REVI[xs; (x . acc)]
```

図 1.4: CPS 変換規則

```
void func(continuation_t cont, object obj) {
  if (スタックの長さが閾値を超えた) {
    resume* r = malloc(適切なサイズ);
    r->fn = func;
    r->var[0] = cont;
    r->var[1] = obj;
    スタック上の必要なものをヒープにコピー;
    longjmp(entry_point, r);
  }
  func の本来の処理
}
```

図 1.5: Cheney on the M.T.A. のアイディア

をどんどん伸ばしていきます。そして、関数の先頭においてスタックの長さを確認して、もしスタックの長さが閾値を超えたら、以下の処理を行います。

1. 今現在の関数とその引数をヒープに退避
2. `longjmp` でスタックを縮める
3. 退避した情報を使って処理を再開

ただこれだけです。関数が呼び出し元に戻ることは無いのですから、スタックに積まれている戻り先は捨ててしまって構わないでしょう。処理の再開に必要なのは、関数と引数ですが、この引数が過去のスタックフレームを参照していると困ります[5]。そこで、`longjmp` でスタックを縮める前に、スタックから必要なものだけをヒープにコピーしてやるのです。この「必要なもの」を探す作業は、後述する Cheney の copying GC のアルゴリズムを使用します。ここまでの考えをコードに直したものを図 1.5 に示します。

1.3.3 Cheney の copying GC

copying GC とは代表的な**ごみ集め**（garbage collection）のアルゴリズムです。ヒープを **from-space** と **to-space** の 2 つに等分して、まずは from-space だけにオブジェクトを割り当てていきます。from-space が満杯になると、「現在使用中のもの」だけを to-space にコピーして、「今後二度と使わないもの」は from-space に置き去りにします。そして from-space と to-space の役割を反転させ、再び（新たな）from-space にオブジェクトを割り当てていきます。「現在使用中のもの」とは、グローバル変数やレジスタなど、必要なものの集合である**ルートセット**に含まれるオブジェクトと、それらのオブジェクトから参照されるオブジェクトです。そのため、ごみ集めの際には、まずルートセットの中身を見て、もし from-space を参照している場合は、それを

[5]例えば、引数がペア（コンスセル）で、その CAR/CDR 部が過去のスタックフレームを参照しているというケースが考えられます。

```
void copyingGC() {
  int i;
  object *tmp = from_space, *p = to_space;
  free_ptr = to_space;
  for (i = 0; i < rootset_size; ++i)
    rootset[i] = copy(rootset[i]);
  while (p < free_ptr) {
    for (i = 0; i < p->num_children; ++i)
      p->child[i] = copy(p->child[i]);
    p += p->obj_size / sizeof(object*);
  }
  from_space = to_space;
  to_space = tmp;
}
object* copy(object* p) {
  if (to_space <= p && p < to_space_end) {
    return p;
  } else if (p->type != TYPE_FORWARDING) {
    memcpy(free_ptr, p, p->obj_size);
    p->type = TYPE_FORWARDING;
    p->forwarding_pointer = free_ptr;
    free_ptr += p->obj_size / sizeof(object*);
  }
  return p->forwarding_pointer;
}
```

図 1.6: Cheney の copying GC

to-space にコピーします。そして、コピーしたオブジェクトが from-space を参照している場合は、参照しているオブジェクトも to-space にコピーするというようになります。Cheney さんの copying GC は、コピーの仕方が工夫されており、to-space をキューとして使用することで、ごみ集めの際に、余分なスペースを使わずに copying GC を行えるアルゴリズムです。図 1.6 に実装例を示します。

さて、ここで Cheney on the M.T.A. のアルゴリズムに話を戻します[6]。`longjmp` でスタックを縮める前に、必要なものをヒープにコピーする必要があるのでした。このコピーの作業は、スタックを from-space、ヒープを to-space とみなすと、copying GC だと考えることができるのです[7]。この Cheney on the M.T.A. のアイディアを使うことにより、Scheme のプログラムを C に変換することができます。ようやく私のはじめての Scheme コンパイラを書く準備が整いました。

1.4 実装

世の中ではじめて Cheney on the M.T.A. のアイディアを使用した Scheme の処理系は Chicken Scheme（以下 CHICKEN）という処理系です [5]。そこで私はこの CHICKEN に敬意を払って、**Kashiwa Scheme**（以下 KASHIWA）という名前の処理系を作ることにしました[8]。

1.4.1 処理の概要

Cheney on the M.T.A. のアイディアを使うためには、まず Scheme のプログラムを CPS 変換するのですが、Scheme のすべての構文に対応した CPS 変換を行うのは、なかなか大変な作業です。そこで、CPS の更に前に準備をすることにします[9]。Scheme には様々な構文がありますが、これらをすべて「定数」「変数」「手続き呼び出し」といった基本的な構文と、`if`、`set!`、`quote`、`lambda` のみを使ったプログラムに変換します。そうす

[6] ここまで読んでお分かりかと思いますが、Cheney on the M.T.A. の Cheney とは、copying GC のアルゴリズム（の一種）を考えた人の名前です。では残りの “on the M.T.A.” は何かといいますと、1959 年に “Charlie on the M.T.A.” という歌があったそうで、この歌のタイトルのもじりのようです。この歌の “Oh, will he ever return? No, he’ll never return,” という歌詞が CPS 変換された関数を連想するので、このようなタイトルにしたのだと思います。
[7] 普通は from-space と to-space は同じ大きさにするのですが、常に to-space の方が大きいのであれば問題は起こりません。
[8] 関西で「かしわ」というと鶏肉を意味します。
[9] 鶏肉を美味しくいただくためには、下味を付けるのが大事ですから。

```
CPS[(set! var atom); cont] = (cont (set! var atom))
CPS[(op arg1 ... atom argN ...); cont] =
  CPS-EVLIS[REV[(op arg1 ... argN ...)];
            REV[(f x ... y ...)];
            (f cont x ... atom y ...)]
```

図 1.7: CPS 変換の追加規則

ると、CPS 変換はこれらの構文だけを考えればいいことになります。cond や let は、if や lambda に展開されるマクロとして実装するわけです。このため、コンパイルの処理は下記のように 3 段階に行われます。

構文の変換（macro expand） ⇒ CPS 変換 ⇒ C 言語に変換

CPS 変換された Scheme のプログラムを C のプログラムに変換する作業は意外と簡単です。既に述べたとおり、CPS 変換後には Scheme の構文はわずかな種類しか登場しないので、それぞれに対応した C のプログラムを生成するコードを力技ですべて書いてやればいいのです。

1.4.2 手続きの表現

Scheme の手続きは first-class のオブジェクトであり、自由に扱うことができます。一方、C でも関数ポインタは自由に扱えるので、関数ポインタを使えば手続きを表現できそうですが、少し問題があります。Scheme では入れ子手続きが許されており、内側の手続きは外側の手続きの変数にアクセスすることができます。いわゆるクロージャというやつです。つまり、手続きとは、処理と環境（外側の変数の情報）の組と考えることができます。そこで、関数ポインタと環境の情報を持つ構造体を作り、それを用いて手続きを表現することにしました。

1.4.3 CPS 変換の小技

CPS 変換を用いると、大量の手続きが生まれます。これらすべての手続きは C の関数に変換され、すべての手続呼び出しは C の関数呼び出しに変換されます。関数呼び出しが多くなると、処理速度が遅くなります。また、スタックの成長速度は速くなるため、スタックからヒープへのコピーが頻発するようになります。そのため、手続き呼び出しの数は少ないに越したことはないのです。

CPS 変換を少し工夫してやると、手続きの数を減らすことができます。例えば、set!の式を CPS 変換すると、第 2 引数を評価してから、実際に代入を行う手続きを呼び出します。しかし、第 2 引数が手続き呼び出しを行わない場合は、直接代入しても問題がないわけです。また、手続き呼び出しにおいても、実引数を 1 つずつ評価して手続きを呼び出していますが、これも、実引数に手続き呼び出し以外の式がある場合は、処理を簡略化できます[10]。これらを実現するための追加の変換規則を図 1.7 に示します。

1.4.4 スタックに対するごみ集め

スタックが延びてきたら、スタックに対してごみ集めを行い、longjmp でスタックを縮めるという処理を見ていきます。まずは、KASHIWA が生成したコードである図 1.8 を御覧ください。関数の先頭で check_stack という関数を呼んでいます。check_stack はスタックの深さを確認して[11]、もしスタックが閾値より長くなっていれば、引数として渡された情報（この場合 fun112_など）をヒープに退避します。次に、グローバル変数と退避した情報をルートセットとして、ごみ集めを行います。これが完了すると、longjmp でスタックを縮め

[10] ただし、これをすると実引数の評価順序が統一されなくなります。しかし、Scheme は実引数の評価順序を定めていないため問題がないのです。

[11] ローカル変数のアドレスを使うと、スタックがどの程度伸びたか確認することができます。

```
static void fun112_(env_t* penv113_, lobject g108_) {
  cont_t clos125_;
  env_t* cenv114_;
  check_stack(fun112_, penv113_, 1, g108_);
  ... (略) ...
}
```

図 1.8: KASHIWA が生成したコード

て、退避した情報を元に、再度 fun112_を呼び出して処理を再開します[12]。しかし、厳密にはこれだけでは問題があります。ヒープからスタックに対する参照がある可能性があるのです。その対策を次節で述べます。

1.4.5 Remember Set

例えば、あるクロージャがスタックからヒープにコピーされ、その後に、クロージャのもつ変数の値を「新しく作った」ペア（コンスセル）に書き換える場合を考えます。新しく作ったペアというのは当然スタックに置かれています。つまり、ヒープ（クロージャ）からスタック（ペア）に対する参照が生まれます。ここで、スタックが延びてきて、スタックに対するごみ集めが起動すると、このクロージャはルートセットに含まれないため、ペアがごみ扱いされて捨てられてしまう可能性があります。

このようなことが起きないようにするために、ヒープからスタックに対する参照はすべて記憶することにします。これを **remember set** と呼びます。ヒープ上にあるオブジェクトに対して代入処理をするときは、代入するものがスタック上にあるかどうか確認して、もしスタック上にある場合は、（ヒープ上の）オブジェクトを remember set に追加します。remember set をルートセットに含めることにより、正しくごみ集めを行うことができます。remember set を正しく作るためには、オブジェクトに対する代入処理を常に監視する必要があります[13]。これは速度のことを考えるとあまり好ましくは無いのですが、Scheme では代入処理を積極的に使うのを嫌う文化あるため、remember set を作ることによるオーバヘッドはあまり気にしなくても良いでしょう。

1.4.6 ヒープに対するごみ集め

何回もスタックからヒープにコピーをしていると、ヒープもあふれてきますので、その場合はヒープに対するごみ集めを行う必要があります。

ヒープは FROM と TO に 2 等分して、copying GC をすることにしました。スタックに対するごみ集めは、スタックを from-space、ヒープを to-space とみなした copying GC でしたが、ヒープに対するごみ集めは、スタック及びヒープの FROM を from-space、TO を to-space とみなした copying GC として書くことができます。つまり、対象となる場所が少し変わっただけでやる仕事はほぼ同じです。ヒープに対するごみ集めを行うために書き加えたコードはわずか 100 行ほどです。すばらしく楽です。

ごみ集めをしてもヒープがあふれる場合は、ヒープを拡大しなければなりません。この処理も本質はシンプルなもので、スタック及び従来のヒープ全体を from-space、新しく確保した大きなヒープを to-space とみなした copying GC を行うことで、オブジェクトの移動やポインタの書き換えを行うことができるのです。細かいところは色々と厄介なのですが、本稿では割愛させて頂きます。

1.4.7 続きは Web で

実装において色々と厄介な問題があります。その問題をどう解決するかというのは、なかなか面白いのですが、紙面の都合上、書くことが出来ません。非常に残念ですが仕方ありません。細かいところを知りたいという奇特な方は、github にソースコードを置いてますので、そちらを御覧ください。

[12] スタックの長さに問題がない場合は処理を続けます。そう、check_stack は return を使うのです。return を使わないというのはあくまでも Scheme の手続きを変換して作った関数の話。それ以外の関数は return しても構いません。

[13] この処理のことをライトバリアと呼びます。

OS	Mac OS X 10.5.8
CPU	Intel Core 2 Duo 2GHz
Memory	DDR3 1067MHz 2GB

表 1.1: ベンチマーク環境

```
;;; tak.scm
(define (tak x y z)
  (if (<= x y)
      z
      (tak (tak (- x 1) y z)
           (tak (- y 1) z x)
           (tak (- z 1) x y))))
(write (tak 18 9 0))
(newline)
```

図 1.9: John McCarthy の tak

http://github.com/zick/kashiwa

1.5 性能評価

KASHIWA はまだまだ作りかけの未熟なコンパイラですが、コアとなる部分はすでに動いています。ということで、簡単なベンチマークプログラムを走らせて、他の処理系と性能を比較して見ることにしました。ベンチマーク環境は表 1.1 の通りです。ベンチマークに使ったプログラムは、図 1.9 の tak 関数です。競争相手に選んだのは、KASHIWA と同じく C へのコンパイルを行う CHICKEN と、中間コードインタプリタ方式をとっている Gauche[6] です。また、参考までに私が 1 回生の時に書いたインタプリタ murasaki でも計測をして見ました。結果は表 1.2 の通りです。

残念ながら、CHICKEN や Gauche には負けてしまいました。でも、ちゃんと勝負になっている感じです。murasaki の実行時間を見てください。**これは誤字ではありません**。これを見ると KASHIWA が相対的に非常に優れた処理系であるような気がしてきます。気がするだけかもしれません。

1.6 おわりに

「Lisp と、Lisper に幸あれ」

参考文献

[1] 湯浅 信吾. はじめての Scheme インタプリタ. Lime Vol.34, 2006.

Implementation	time (sec)
CHICKEN	1.16
Gauche	1.48
KASHIWA	2.37
murasaki	3282

表 1.2: tak.scm の実行時間

[2] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised5 Report on The Algorithmic Language Scheme. ACM SIGPLAN Notices, 33(9), September 1998.

[3] Henry G. Baker. CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. (`http://www.pipeline.com/~hbaker1/CheneyMTA.html`)

[4] 林 奉行. Common Lisp に継続を. Lime Vol.42, 2010.

[5] FAQ - The Chicken Scheme wiki. (`http://wiki.call-cc.org/man/4/faq#why-yet-another-scheme-implementation`)

[6] Gauche - A Scheme Implementation. (`http://practical-scheme.net/gauche/`)

2 Forth を使え

京都大学大学院 情報学研究科 修士 1 回 小宮山敦史

2.1 概要

この記事では Forth ぽいものを C 言語で実装する。ほぼ LetOverLambda[1] の影響のみを受けているため、記法などがよくある Forth 実装と異なる箇所がある。対応表を 2.4 節に記載する。詳細は Wikipedia[2] に譲る。ソースコード本体は `http://github.com/Atsushi-KOMIYAMA/c_forth` においてある。ここでは、エラーチェックなどは省いてあったり、行数節約のために分けて書いてあったものを統合していたりする。

2.2 Forth の実装

まず Forth オブジェクトを定める。とりあえずは数値型、シンボル型を考える。残りは組み込み関数および、スレッドへのポインタであるが、詳細は後述する。拡張などのためにこれらは `set_num`，`get_num` などの操作関数を用いて、直接アクセスしないようにする。格納しているものを判別するためには下位 bit を型タグとして利用する[1]。

```
typedef union forth_obj{
  int num;
  char* sym;
  forth_obj* (*prim)(void);
  forth_obj* thread;
} forth_obj;
```

2.2.1 データ構造

データを格納するものは `pstack`、`rstack`、`dictionary`、`threaded_code`、`pc`、`compiling` の 6 つある。

まず `pstack` と `rstack` だ。`pstack` はパラメータスタック[2]と呼ばれ、サブルーチンにパラメータを受け渡しするのに用いる。`rstack` はサブルーチン呼び出しの履歴を保存するリターンスタックである。操作関数として `push`，`pop`，`peek` あたりを定義しておく。

次に `dictionary` である。サブルーチンを Forth ではワードと呼び、`dictionary` はそのワードの集合で構成される。要は現在呼び出せるサブルーチンの塊と思えばよい。

```
typedef struct forth_word{
  char* name;
  int immediate;
  forth_obj* thread;
} forth_word;
forth_word dict[FORTH_DICT_SIZE];
int dict_index = 0;
```

[1] 32bit 環境では下位 2bit を型タグにする。数値型は 30bit しか使えないし、組み込み関数も 4byte おきに配置する必要がある。
[2] 単にスタックとよばれることもある [2]。

１つのワードは名前、immediate かどうか (後述)、本体へのポインタで構成できる。

次は thread_code である。これはワードの本体を格納する領域と考えればいいだろう。

大事な項目なので詳細は 2.2.2 節で説明する。

pc はプログラムカウンタで、実行中のコードを指す。スレッドへのポインタを使う。

最後に compiling である。これは Forth インタープリタの状態を表す。これも詳細は 2.2.6 節で説明する。

2.2.2 スレッデッドコード

ここでいうスレッド (thread) とは並列処理のそれと異なり、コードのコンパイルというメタプログラミングについて語る枠組みである。スレッディング (threading) とはコードをメモリに組み込む過程であり、Forth はこの過程を制御できる。Forth の特徴はパッと見ではスタックだが、実際はこのスレッドが Forth を Forth たらしめている。

本記事では間接スレッデッドコード (threaded code) という方法で実装する。

Forth オブジェクトは組み込み関数や他のオブジェクトへのポインタを持てるようにしたので、ワードの本体は Forth オブジェクトの列で構成できる。ワードの区切りのための終端記号を用意すれば、巨大な配列１つでまかなえる。

```
forth_obj forth_thread_code[FORTH_THREAD_SIZE];
int thread_index = 0;
```

```
threaded code
+----+----+----+----+--
|obj1|obj2|term|obj3|
+----+----+----+----+--
   ^              ^
   |     +--------+
+--+--+--+--+----------
|  .  |  .  |
+-----+-----+----------
dictionary
```

2.2.3 インタープリタの動作

main 関数からトップダウンに説明する。まずスタックの初期化を行う。次にプリミティブなワードを定義する。その処理としては、まず forth_install_prims で組み込み関数を定義し、そして forth_install_stdlib でブートストラップコード、つまり Forth で書かれたワードを定義する。これらについてはすぐに説明する。そのあとスペースで区切られた 1 単語 (ワード) ごとに eval_forth で評価する、対話モードに移行する。

```
int main(void){
  char v[BUF_SIZE], *ret, *last;
  init_stack(&rstack);
  init_stack(&pstack);
  forth_install_prims();
  forth_install_stdlib();
  while(1){
    fgets(v, BUF_SIZE, stdin);
    for(ret = v; ret != NULL; ret = last){
      ret = strtok_r(ret, " ", &last);
      eval_forth(ret);
    }
  }
  return 0;
}
```

2.2.4 組み込み関数の定義

組み込み関数を forth_install_prims や add_foth_prim を用いて dictionary に定義する。具体的な方法はソースコードを参照して欲しい。

各組み込み関数として、何もしない nop、パラメータスタックの上から 2 つを掛け算する*、パラメータスタックのトップを捨てる drop の他、パラメータスタックのトップを複製する dup、パラメータスタックの上から 2 つを入れ替える swap、パラメータスタックのトップを表示する print などを定義する。

pc を組み込み関数側でインクリメントしていることに注意して欲しい。分岐などを組み込み関数で実装するために、全組み込み関数で pc を制御する方式を取った。

```
void F_nop(void){
  pc++;
}
void F_mul(void){/* * */
  forth_obj ret;
  set_num(&ret, get_num(pop(&pstack)) * get_num(pop(&pstack)));
  push(ret, &pstack);
  pc++;
}
void F_drop(void){
  pop(&pstack);
  pc++;
}
/* 略 */
```

2.2.5 ブートストラップコードの定義

forth.lib というファイルにテキストで Forth プログラムを記述し、Forth ワードを定義することにする。

```
#define FORTH_LIBRARY_NAME "forth.lib"
void forth_install_stdlib(void){
  FILE* fp = fopen(FORTH_LIBRARY_NAME, "r");
  int ret;
  char sym[FORTH_WORD_NAME_SIZE];
  do{
    ret = fscanf(fp, "%s", sym);
    if(ret == 0 || ret == EOF){ break;}
    eval_forth(sym);
  }while(1);
  fclose(fp);
}
```

2.2.6 ワードのコンパイル

ここで eval_forth の説明の前に、compiling という変数とワードの属性 immediate について説明する。これらはワードがコンパイルされるか実行されるかどうかを表す。ここでいうコンパイルは、機械語への翻訳のことではなく、中間言語への翻訳である。ただし、中間言語がポインタなのであまりコンパイルという感じがしないかもしれない。

compiling、immediate と実行されるかコンパイルされるかの関係を表 2.1 に示す。

ここでそのコンパイル中かどうかを切り替えるワードを定義する。

```
void F_l_bracket(void){/* [ immediate */
  compiling = FALSE;
  pc++;
}
void F_r_bracket(void){/* ] */
  compiling = TRUE;
  pc++;
}
```

表 2.1: compiling，immediate とワードが実行されるかコンパイルされるかの関係。

	immediate == FALSE	immediate == TRUE
compiling == FALSE	ワードが実行される	ワードが実行される
compiling == TRUE	ワードがコンパイルされる	ワードが実行される

[はコンパイルをオフにするため、常に実行できるよう immediate にする。

2.2.7 eval の定義

eval_forth はワードを受け取り以下のような処理を行う。

1. ワードが dictionary にあれば、
   (a) compiling == TRUE かつ immediate == FALSE の時、現在コンパイル中のスレッド (通常 dictionary 中で最も新しく作られたワード) にそのワードがコンパイルされる。
   (b) それ以外の時、そのワードが探索されて実行される。
2. dictionary にない時、まず先頭の一文字を見る。
   (a) クォート (') で始まる場合、
      i. compiling == TRUE なら、クォートを除いた部分をコンパイルする。
      ii. compiling == FALSE なら、クォートを除いた部分をパラメータスタックにプッシュする。

      この処理は、シンボルを明示的にパラメータスタックにプッシュするためにある。クォートなしのシンボルを eval に与えた時に、dictionary にない場合パラメータスタックにプッシュされることにすると typo の判別が難しいためである。現在の実装ではクォートなしのシンボルはエラーにしている。
   (b) コンマ (,) で始まる場合、そのワードをコンパイルする。これは immediate であるワードを強制的にコンパイルするために使う。
   (c) それ以外の場合、数であるはずなのでその数をパラメータスタックにプッシュする。

コンマやクォートで始まるワードが dictionary にない時の処理にあるのは、それらのワードは dictionary に登録しないからである。

forth_inner_interpreter はワードが実行される挙動を示す。

```
void forth_inner_interpreter(void){
 do{
   if(pc_end_p(*pc)){
     pc = get_thread(pop(&rstack));
   }else if(prim_p(*pc)){
     exec_prim(pc);
   }else if(thread_p(*pc)){
     forth_obj tmp;
     set_thread(&tmp, pc + 1);
     push(tmp, &rstack);
     pc = get_thread((forth_obj)pc->thread);
   }else{
     push(*pc, &pstack);
     pc++;
   }
 }while((!pc_end_p(*pc)) || (rstack.index != 0));
}
```

プログラムカウンタ pc が組み込み関数を指していればそれを実行する。他のスレッドを差していればワードの呼び出しを意味し、pc の値をリターンスタックにプッシュして新しいスレッドにジャンプする。そうでなければ、それをパラメータスタックにプッシュする。この内部インタープリタはスレッドの終端に到達し、リターンスタックに復帰するべきスレッドがなければ終了する。

ここまで作ればとりあえず動くものが出来る。

2.2.8 ワードの定義

この時点では、インタープリタとして動作するものの、プログラミング言語としての機能は少ない。新しいプリミティブ関数を用意する。

新しいワードを作る `create`、ワードに名前をつける `name`、ワードを即値に設定する `immediate` を定義する。

これらを用いれば、新しいワードを自分で定義できる。

例えば、

`create ] dup * [ 'square name`

で `square` を定義し、

`3 square print`

で 9 が表示される。という事ができるようになる。

とは言うものの、毎回 "`crete ]` 何とか `[` 名前 `name`" を入力するのは面倒なので、省略できるよう新しいワード "}" と "{" を作る。

```
(Forth.lib)
create ] create ] [ '{ name
{ ,[ [ '} name immediate
```

プリミティブ関数は安易に増やしたくないし、幸いこれらは今までのプリミティブ関数の組み合わせで作れるので、`Forth.lib` に記述することにする。これで、

{ `dup *` } `'square name`

と短く書けるようになる。

2.2.9 条件分岐を作る

プログラミング言語を名乗るなら if くらいないと[3]いけないということで `if` 系を作る。

以下のような仕様を考える。`if A then` でパラメータスタックのトップが偽なら `then` まで制御がジャンプする。`if A else B then C` なら、条件で A か B を実行し、その後どちらの場合でも C を実行する。ここで、A、B、C は 1 つ以上のワードである。

まず準備としてプリミティブ関数 `branch-if` を作る。これはパラメータスタックのトップが 0 なら次のスレッドを飛ばす。そうでないなら次のスレッドに制御を移す。リターンスタックに pc をプッシュしていないので、そのスレッドが終わったら、`branch-if` を呼んだ元に戻ることになる。

```
void F_branch_if(void){
  pc = (get_num(pop(&pstack)) != 0) ? get_thread(pc[1]) : pc + 2;
}
```

[3] ループも欲しいが紙面の都合上割愛する。

次に!(store)、compile と here を用意する。!はパラメータスタックの上から 2 つ目の場所を指すようにトップが指すスレッドを設定する。compile は次のワードをコンパイルする。here は直前にコンパイルした場所をプッシュする。

```
void F_store(void){/* ! */
  forth_obj *loc = get_thread(pop(&pstack)), *thread = get_thread(pop(&pstack));
  set_thread(loc, thread);
  pc++;
}
void F_compile(void){
  Forth_compile_in(pc[1]);
  pc += 2;
}
void F_here(void){
  forth_obj tmp;
  set_thread(&tmp, Forth_thread_code + thread_index - 1);
  push(tmp, &pstack);
  pc++;
}
```

これらを組み合わせることで if、then、else を作ることができる。

```
{ compile not compile branch-if compile nop here } 'if name immediate
{ compile nop here swap ! } 'then name immediate
{ compile 1 compile branch-if compile nop here swap compile nop here swap ! } 'else name immediate
```

これがどのように動作するか以下の例で見てみよう。

```
{ 0 swap - } 'negate name
{ dup 0 < if negate then } 'abs name
```

まず negate は普通にコンパイルされる。

```
             +-----------------+
  +----------+-------------+   |
  v          v             |   |
----+--   -+---+-   -+---+-+-+-+-+---+----
swap|  ... |-  | ... |0  | . | . |END|
----+--   -+---+-   -+---+---+---+---+----
threaded code         ^
                      negate
```

次に abs のコンパイルを始める。まず、dup、0 < がコンパイルされた後

```
      swap  -      dup      <
        ^   ^       ^       ^
--+---+-+-+-+-+---+-+-+---+---+--
  |0  | . | . |END| . |0  | . |
--+---+---+---+---+---+---+---+--
   ^                threaded code
   negate
```

if の中身がコンパイルされ、パラメータスタックに nop の場所がプッシュされる。

```
      swap  -      dup      <  not branch-if
threaded^   ^       ^       ^   ^   ^   nop
code    |   |       |       |   |   |   ^
--+---+-+-+-+-+---+-+-+---+-+-+-+-+-+-+-+-+--
  |0  | . | . |END| . |0  | . | . | . | . |
--+---+---+---+---+---+---+---+---+---+---+--
   ^               ^                    ^
   negate          abs                  |
                                        |
pstack          +-----------------------+
--------------+-+-+---
              | . |
--------------+---+---
                ^
                top
```

negate、then の中の nop がコンパイルされ、パラメータスタックに nop の場所がプッシュされる。

```
       swap                     branch-if  negate
threaded^   -      dup      <  nop  ^   nop ^   nop
code    |   ^       ^       ^   ^   |   ^   |   ^
--+---+-+-+-+-+---+-+-+---+-+-+-+-+-+-+-+-+-+-+-+-+--
  |0  | . | . |END| . |0  | . | . | . | . | . | . |
--+---+---+---+---+---+---+---+---+---+---+---+---+--
   ^               ^                    ^       ^
   negate          abs                  |       |
                +-----------------------+       |
pstack          |   +---------------------------+
--------------+-+-+-+-+--
              | . | . |
--------------+---+---+--
                ^
                top
```

パラメータスタックの上 2 つを入れ替えて

```
       swap                     branch-if  negate
threaded^   -      dup      <  nop  ^   nop ^   nop
code    |   ^       ^       ^   ^   |   ^   |   ^
--+---+-+-+-+-+---+-+-+---+-+-+-+-+-+-+-+-+-+-+-+-+--
  |0  | . | . |END| . |0  | . | . | . | . | . | . |
--+---+---+---+---+---+---+---+---+---+---+---+---+--
   ^               ^                    ^       ^
   negate          abs                  |       |
                    +-------------------+       |
pstack          +---+---------------------------+
--------------+-+-+-+-+--
              | . | . |
--------------+---+---+--
                    ^
                    top
```

ストアする

```
      swap  -      dup           not branch-if
        ^   ^       ^           ^   ^
                                            negate
threded |   |       |       <   |   |       ^   nop
code    |   |       |       ^   |   |       |   ^
--+---+-+-+-+-+---+-+-+---+-+-+-+-+-+-+---+-+-+-+-+---+--
  |0  | . | . |END| . |0  | . | . | . | . | . | . |END|
--+---+---+---+---+---+---+---+---+---+-+-+---+---+---+--
   ^               ^                    |       ^
   negate          abs                  +-------+
```

else を使う例は以下のようになる。

```
{ evenp if 0 else 1 then } 'mod2 name
```

先程と同様に考えていけば、上手く行っていることがわかるはずだ。

2.3 最後に

[1] を読んで実装したのだが、あとから [2] や [3] を見るとインタープリタの状態は `compiling` だけではないようだ。素直に `state` あたりの名前にしておけば拡張しても違和感がないだろう。そのうちの更新で変更されるかもしれない。

Forth と共にあらんことを。

2.4 対応表

本記事の実装と一般的な実装のワードの名前の違いなどの対応表を表 2.2 に示す。

表 2.2: ワード名などの対応表

	本記事の実装	一般的な実装
パラメータスタックのトップを表示	`print`	.
ワード定義開始	{	:
ワード定義の終了	}	;
ワードの名前付	ワードを作ってから 新しいワードを作るまでに *name* `name`	`:` の直後に *name*
コメント	なし	(*comment*)
強制コンパイル	シンボルにコンマ (`,`) をつける	`postpone` *name*
`here` が返す場所	最後にコンパイルしたスレッド	次にコンパイルするスレッド

参考文献

[1] ダグ・ホイト, 株式会社タイムインターメディア HOP プロジェクト 訳, LET OVER LAMBDA

[2] Forth - Wikipedia, `http://ja.wikipedia.org/wiki/Forth`

[3] David J. Nordstrom, Threading Lisp

3 電子工作のすすめ (回路素子編)

電子システム工学課程 4 回 軸原 健太

3.1 はじめに

コンピュータ部には大まかにいってプログラムを書く人と電子工作をする人がいます。しかし、”大学に入るまでに電子工作をしたことがある” という人はほぼ皆無なのではないでしょうか。

そこで今回は、電子工作を始めるにあたって必要な知識のうち、”どんなパーツ (回路素子) を使うのか？”に絞って説明します。この知識を最初の核として、どんどん回路をつくっていきましょう！ 電子システム工学課程の人なら、2 回、3 回生での電子回路実験のために早めに練習しておくのもよいと思います。

3.2 回路素子は分けると 2 種類

電気/電子回路を構成する素子は、受動素子/能動素子の二種類に大別されます。受動、能動という言い方は”供給された電力を変化させるか否か” という観点からのようです。また、”電流電圧特性が直線か否か (グラフが線形か非線形か)” という観点でもあるようです。

3.2.1 自分自身は動かない受動素子

受動素子には主に次の 3 種類があります。

- 抵抗器
- キャパシタ[1]
- コイル (インダクタ)

これだけ。まずはこの 3 つを覚えておけば基礎は OK です。

まさに”抵抗” する素子 ”抵抗器”

最初に出てくるのは抵抗器です。これがない回路などないといってもいいでしょう。電圧と電流の関係式はおなじみ” オームの法則” で表され、それは比例定数 R(単位：Ω(オーム)) を使って次式のように表されます。

$$v(t) = Ri(t)^2 \tag{3.1}$$

用途は単純、”電圧降下を起こし電流を制限する”。もし回路中に抵抗器がなければ、大電流が流れてしまい回路素子は一瞬で破壊されることとなります。

[1]” コンデンサ” という呼び方は日本でのみ通用します
[2]小文字は” これは時間関数だ” ということを、大文字は”定数だ” ということを表すという慣習があります

電荷という水を貯めるバケツ ”キャパシタ”

抵抗器の次に使われるのがキャパシタです。これは 2 つの導体板を近づけておいたような形になっていて、電荷を導体板の両端に貯めておくことができます。電圧と電流の関係式は比例定数 C(単位：F(ファラド)) を使って次のように表します。

$$i(t) = C\frac{dv(t)}{dt} \tag{3.2}$$

この素子は交流のときに意味を持ちます。直流に対しては”回路がぶつ切りされている”のと同じです。[3]
このキャパシタを何に使うのかというと、

- 信号の直流成分を取り除く
- 電源と一緒に使い電源電圧の変動を避ける[4]
- コイルと一緒に共振回路を作る

などがあります。

変化したがらない素子 ”コイル”

先ほどの 2 つよりは使用頻度が落ちますが、それなりに使います。形は導線をバネのように巻いたものです。電圧と電流の関係式は比例定数 L(単位：H(ヘンリー)) を使って次のように表します。

$$v(t) = L\frac{di(t)}{dt} \tag{3.3}$$

これも交流のときに意味を持ちます。直流に対しては”導線が巻かれている”のと同じです。[5]
このコイルを何に使うのかというと、

- 変圧器 (電圧を変化させる)
- キャパシタと一緒に共振回路を作る

などがあります。抵抗、キャパシタ、コイルの回路記号とこれらの関係を図 1 に示します。

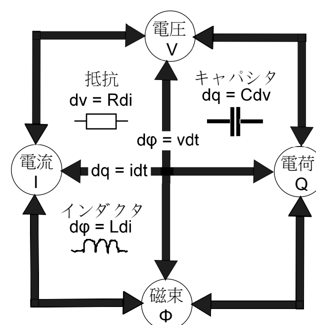


図 3.1: 受動素子の関係

[3]スイッチを入れて十分時間がたったあと (定常状態) での話です
[4]バイパスコンデンサ (パスコン) といいます
[5]これも定常状態での話です

さて、図 3.1 を見てもらうと”磁束”という耳慣れない単語が出てきます。これは、”ある閉曲線の中を貫いた『磁束線』の数”だと考えてください。磁束線というのは、棒磁石の近くに砂鉄を置いたときにできる線のようなものに似ています。(正確にはこれは磁力線といい、磁力線と磁束線の違いは図 3.2 に示します)

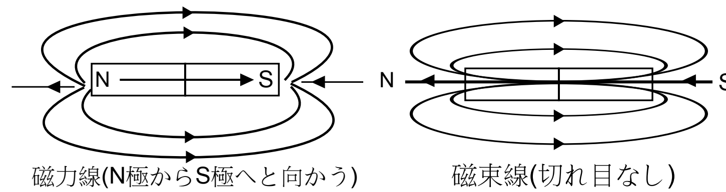


図 3.2: 磁力線と磁束線の違い

また、図 3.1 の右下が空欄になっていますが、対応する素子はちゃんとあります[6]。ただし、現在実用化されていないので割愛しました。もしかしたら 10 年後には日常的に使っているかもしれません。

3.2.2 幅広い電子工作には不可欠な要素、能動素子

次に説明するのは能動素子です。これがないと電力を増幅したり、整流したりすることができません。主だった能動素子は次の 4 つです。

- ダイオード
- トランジスタ
- オペアンプ
- IC

この 4 つは電子回路のあちこちに出てきます。

ここから先は一方通行”ダイオード”

ダイオードは”整流”をさせることができます。要するに、

「あっちからこっち[7]へは電流を通すが、こっちからあっちには電流を流さない」

ということです。図 3.3 に、ダイオードの回路記号を示します。

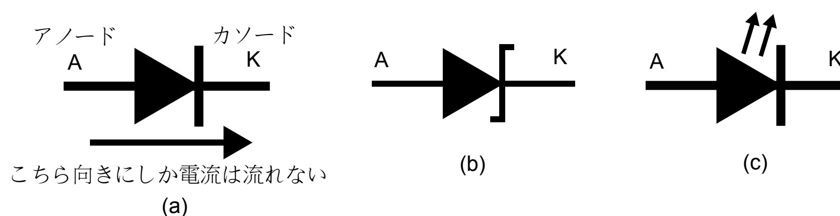


図 3.3: ダイオードの回路記号 (a) ダイオード (b) ツェナーダイオード (c)LED

[6]”メモリスタ”といいます

[7]あっち (A) からこっち (K)、アノード (A) からカソード (K) へという対応を意識

ダイオードは電圧をかけなければ動作しません。電圧のかけ方は次に 3 通りです。

1. アノード側の電圧がカソード側の電圧より高い[8](順方向バイアス)
2. カソード側の電圧がアノード側の電圧より高い (逆方向バイアス)
3. 非常に大きな逆方向バイアスをかける

この 3 通りの電圧のかけ方に対する応答は次の 3 通りです (番号には対応関係にあります)。

1. 電流が流れる。
2. 非常にわずかな電流が流れる (10^{-15}〜$^{-12}$ アンペアのオーダー)。
3. 大電流が流れる (ツェナー降伏、アバランシェ降伏)

3 番目はある種異常な応答ですが、この現象を利用して”一定の電圧降下を起こす”という使い方をする場合があります。そのような使い方をするダイオードを”ツェナーダイオード”といいます。(図 3.3(b))

他にも”電流が流れると発光する”というダイオードもあります。これは”発光ダイオード (LED)”といい、これもよく使われます。(図 3.3(c))

3.2.3　電力増幅はおまかせ”トランジスタ”

トランジスタのうち、よく使われるものに”バイポーラトランジスタ”と”MOSFET”があります。使い方は同じですが、小さくしやすく消費電力が小さいという利点から MOSFET のほうが多用されます[9]。

トランジスタは”3 端子素子”といい、その名の通り、端子を 3 つ持ちます。端子の名前は、バイポーラトランジスタでは”ベース (B)、エミッタ (E)、コレクタ (C)”といい、MOSFET では”ゲート (G)、ソース (S)、ドレイン (D)”といいます。また、トランジスタには”n 型”と”p 型”があり、それぞれ電荷の運び手 (キャリア) が違います (電流の向きが逆になります)。

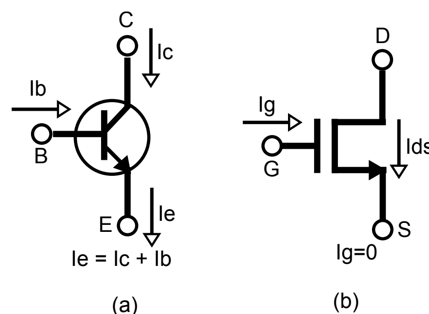


図 3.4: n 型トランジスタの回路記号と電流の向き (a) バイポーラ Tr. (b)MOSFET

[8]0.6V から 0.7V 程度の電位差を要求します
[9]バイポーラトランジスタの利点は”電圧利得 (入出力電圧の比) が大きい””動作が速い”などがあります

使いどころはおおむね次の 3 つです。

- 電力増幅
- スイッチング回路
- 論理ゲート回路

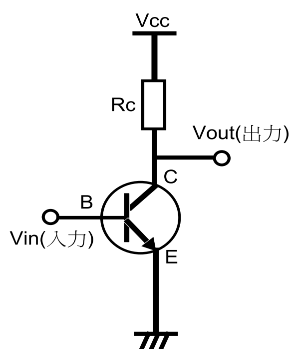


図 3.5: 電力増幅回路

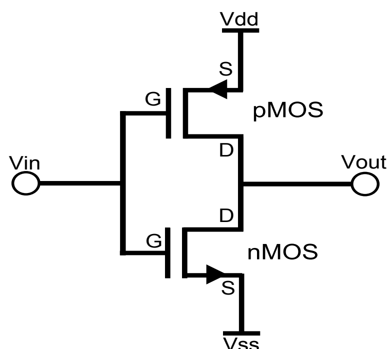


図 3.6: 論理ゲート回路 (インバータ)

3.2.4 強力なユーティリティプレイヤー"オペアンプ"

オペアンプとは 2 つの入力と 1 つの出力をもった"増幅器"です。その点で言えばトランジスタの上位に属するともいえます。しかしオペアンプの用途はそれだけにとどまらず、

- コンパレータ (入力した 2 つの電圧を比較し、対応した出力を行う)
- 積分回路
- 発振回路

など多種多様に及びます。あまりに多いため回路図の例を描くとページが埋まってしまうので、ここでは一部のみ説明します。

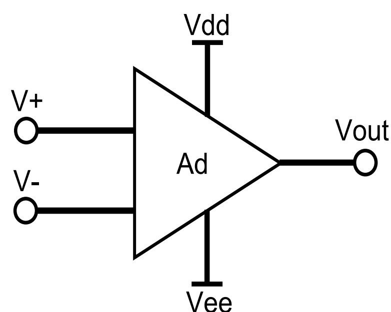


図 3.7: オペアンプの回路記号

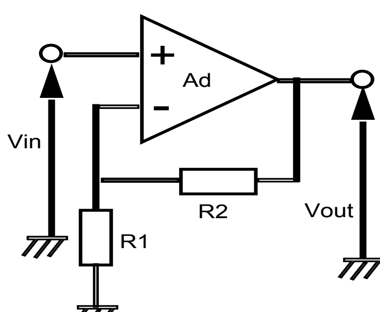


図 3.8: 非反転増幅回路

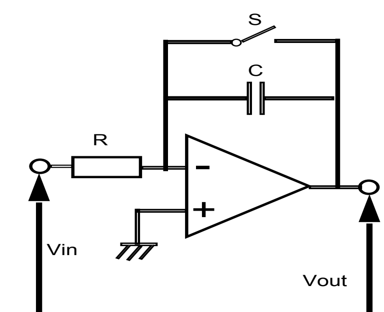


図 3.9: 積分回路

図 3.7 について補足すると、入力は V_+ と V_- の 2 つで、出力は V_{out} です。出力電圧 V_{out} は

$$V_{out} = \begin{cases} V_{dd} & (V_+ \text{が} V_- \text{よりも大きい}) \\ V_{ee} & (V_- \text{が} V_+ \text{よりも大きい}) \\ A_d \times (V_+ - V_-) & (V_+ \text{と} V_- \text{がほぼ等しい}) \end{cases}$$

となります。A_d は"差動利得"といい、実際のオペアンプでは 10^5 から 10^7 程度という大きな値をとります。

よって、オペアンプをそのまま使うと電圧が飽和してしまう[10]のですが、図 3.8 や図 3.9 のように、"入力端子と出力端子を接続する (負帰還といいます)"と、電圧利得を自分で設定することができます。[11]

負帰還がかかっている状態のオペアンプには、2 つ重要な現象が起こっています。それは

- 入力端子には電流が流れない
- 入力端子間の電位差はほぼ 0、すなわち $V_+ = V_-$(これを"ヴァーチャルショート"といいます)

です。これらは、負帰還がかかったオペアンプのある回路を解析するときに重要な概念です。

3.2.5 一つのところに多くが集まる"IC"

最後に紹介するのは"IC(Integrated Circuit/集積回路)"です。これは今まで説明したような"回路素子"というよりは"何らかの目的のために回路素子が多数詰め込まれている"ものです。

ですから用途は非常に多彩というわけですが、その中から名前が付いているものを少し選んでみると、

- 汎用ロジック IC
- 汎用メモリ
- アナログ-デジタル変換回路
- 半導体センサ

などなど。この他にも多数の用途に使われます。

3.3 おわりに

ここまで電子回路制作に必須な回路素子について説明してきましたが、これらはいわゆる"知識のための知識" といった感じで、実際に回路を組んでいくにはまだ多くのことを学ぶ必要があると思います。そのためにはやはり、自分で手を動かして経験していく必要があると思います。

参考文献

[1] 松澤昭,『基礎電子回路工学-アナログ回路を中心に-』, オーム社,2009 年

[2] 電気学会,『電気磁気学基礎論』, オーム社,2005 年

[3] Joseph A. Edminister,『マグロウヒル大学演習 電気回路』, オーム社,2008 年

[4] 菊池正典,『図解でわかる電子回路 トランジスタ、コンデンサから論理･演算回路のしくみまで』, 日本実業出版社, 1999 年

[10] 入力電圧がわずかでも出力電圧はものすごいことになる……わけではなく、V_{dd} 程度が出力されます
[11] 例えば図 3.8 の場合、電圧利得を 2 つの抵抗 R1,R2 で決定することができます

4　昆布鉄道/Nave;train☆ぷろじぇくと！最終章 安全・保護回路の設計

電子システム工学課程 4 回 渡邉 翔一郎

4.1　昆布鉄道/Nave;train☆ぷろじぇくと！

私は 2 回生からコンピュータ部に入部しましたが、その時からずっと学祭では鉄道模型 (N ゲージ) をテーマに電子工作をしてきました (図 4.1)。

鉄道模型とは、実際の鉄道車両や線路、駅などを忠実にモデル化したものです。この鉄道模型では車両を線路の上で電気でコントロールして走らせたり、線路の沿線で踏切や信号などを動作させて楽しむことができます。もう毎年恒例になってしまいましたが (笑)、実写と模型の比較写真を載せておきます (図 4.2)。ぜひ見比べて見てください。

毎年想定外のトラブルや、線路の保守管理の問題が表面化していましたが、改良を重ねていくことでだいぶマシになってきました。今年度で卒業なので、このプロジェクトの締めくくりとして、鉄道模型の安全走行に必要な保護回路について考察してみようと思います。

4.2　安定した回路を設計する

安全のためには、まず回路が安定して動作するというのが前提です。ここでは安定して動作する回路を設計する際の工夫や注意点について説明します。

4.2.1　回路図編

安定して動作させるには、電子部品の性質を把握する必要があります。たとえばバイポーラトランジスタは、スイッチングデバイスとして用いる場合、High 時の電圧として、V_{be} には 0.6V 以上が必要になります。さらに、バイポーラトランジスタは電流制御デバイスなので、I_b の供給元には数十 mA 流せる余裕も必要です。このように、部品の特性を考慮して回路図上のパラメータを決定していきます。

では次の回路図 (図 4.3) を見てください。

これは私が 2 回生の時に設計した、模型のモータを制御する回路です。今回は鉄道模型の安全走行がテーマですので、図の右側にある保護回路に注目します。

保護回路のトランジスタが ON になるのは、V_{be} が 0.6V を超えたときなので、これはベース・エミッタ間にある抵抗が鍵になります。この抵抗で電位差が 0.6V 生じたときにトランジスタが ON になる、つまり抵抗に電流が 3A 以上流れたときに ON になるということです。このトランジスタが ON になればダーリントントランジスタ (ダーリントン接続) のベース電位が 0V になるので出力電流が 0A になります。

図 4.3 では電圧のみ考察していましたが、4.4 の回路ではどうでしょうか。Q3 と Q5 を見てください。2 つのトランジスタをコンプリメンタリに使っています。同様の機能はトランジスタ 1 個でも実現できました。実際、電圧では十分な条件でしたが、電流には余裕がなかったため、トランジスタを 2 個用いて動作を安定させました。

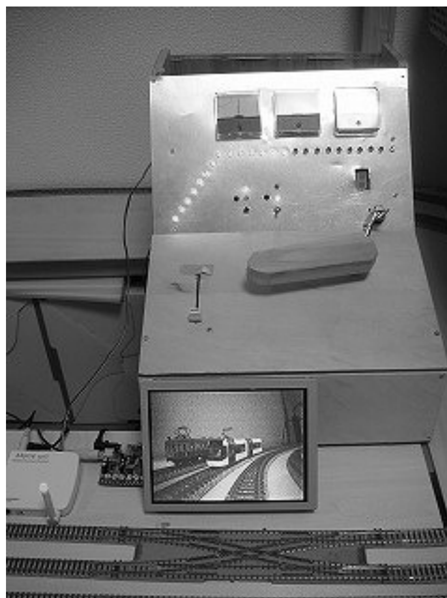


図 4.1: 過去に製作した作品



図 4.2: 実写/模型車両の比較 (７００系のぞみ・実写は京都駅にて撮影)

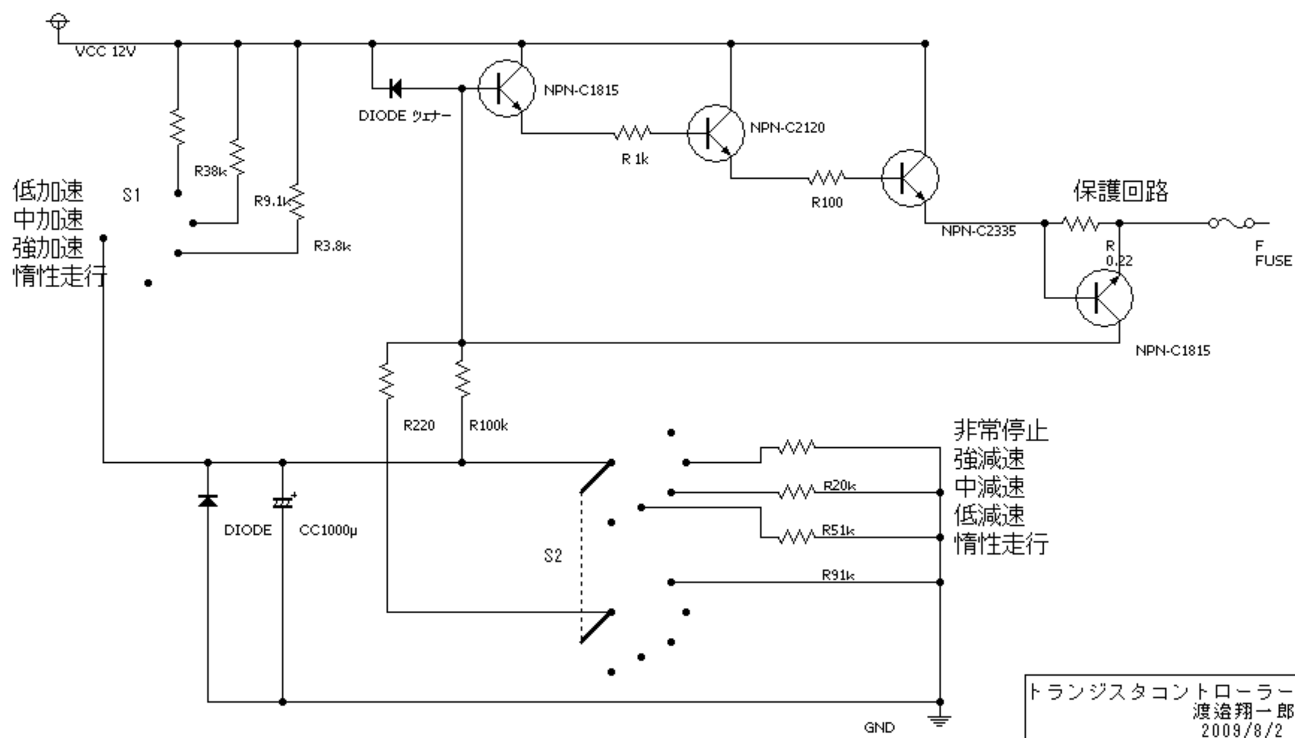


図 4.3: トランジスタコントローラー回路

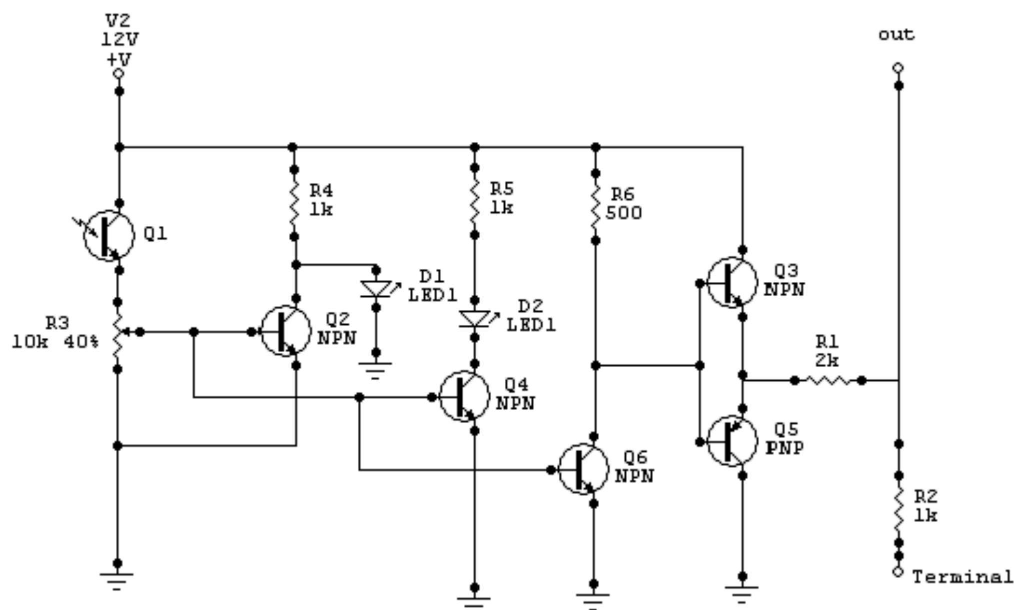


図 4.4: 車両検出＆ D/A コンバータ回路

4.2.2 部品選定編

設計が終わって回路図が書き上がったても、回路図上には現れない電子部品特有のパラメータがあります。それは耐圧、許容電流、許容電力です。トランジスタは型番を選んだ時点でこれらが決まるため回路図上問題ないのですが、抵抗やコンデンサは回路図上にこれらの記述はないため部品を選定する作業があります。図 4.3 の回路で見てみましょう。通常抵抗は 1/4W のカーボン抵抗を使いますが、保護回路では 3A 流れることを前提としていますのでカーボン抵抗では持ちません。実際に電流を流しすぎるとどうなるかというと、抵抗が徐々に焦げていき、煙が出てきます。そうさせないためには、定格電力の大きい抵抗であるセメント抵抗を選ぶ必要があります。また、電解コンデンサについても極性や耐圧制限がありますから注意しましょう。

4.2.3 部品実装編

回路図を設計して、部品も選んで完璧！と思いきや、まだ安定動作には課題があります。図 4.3 では鉄道模型を電圧可変で動かすので、例えば出力を 3V にした場合、電源の 12V との電位差 9V 分の発熱があります。つまり、放熱板や排熱ファンが必要になります。この熱はどこで発生するのか、どこに放熱板をつければいいのかなんて回路図には書いてませんし、理論上では導線などの発熱を考えないので (厳密には回路で電力損失がないと仮定している)、部品を実装するときに考えることになります。というのも、部品の形状にヒントがあったりするからです。

実際、この場面では 2SC2335 で熱が発生するのでこれに放熱板をつけることになります。ちゃんと、2SC2335 には放熱用のネジ穴がついています。

また、導線も許容電流が決まっています。細い線で簡単に配線してしまうと、熱を持ってしまい、絶縁被膜が溶けてしまうことがあります。

4.3 安全な回路を設計する

安全のためには回路の安定動作が前提でしたが、ここではその上で更なる安全性について考えてみます。

4.3.1 定格

まずここで、一般に定格という考え方について説明します。定格とは部品の性能や使用限度のことです。例えば電気の世界で「ダイオードの定格電圧」というと、設計した回路で安定してダイオードを使用できる電圧のことです。定格にはある程度余裕が持たせてありますが、絶対に超えてはならない定格値というのも存在して、それを「絶対最大定格」と呼びます。

4.3.2 抵抗の定格電力

抵抗の W については前節で触れましたが、想定のギリギリで使うのは安全上よくありません。つまり、定格を考えようということです。目安として想定する W 数の 2 倍の W を持つ抵抗をつけることが望ましいとされています。

4.3.3 コンデンサ

コンデンサの耐圧も抵抗同様、定格を考える必要があります。目安として想定する電圧の 2 倍の耐圧を持つコンデンサをつけることが望ましいとされています。

4.3.4 トランジスタ

トランジスタは電流制御デバイスなので、コレクタ、エミッタ、ベース電流ともに定格を守るのはもちろんですが、定格電圧も決まっています。定格電流は注目できていても定格電圧について配慮にかけていることがあります。注意しましょう。

4.3.5 ヒューズ

図 4.3 では最後にヒューズを入れています。これは安定動作には全く関係ない部品ですが、安全性を考えると最大の威力を発揮します。過電流が流れると物理的に導線を切断するので、他の安全回路に比べて誤動作のリスクもなく一番安全です。またヒューズは性質上消耗品ですので、ここまで予算を駆使できないという人にはポリスイッチをおすすめします。ポリスイッチは定格電流より強い電流が流れると素子の抵抗が一気に絶縁体並にまで上昇し、電流を小さくする部品です。これはリセッタブルヒューズとも呼ばれ、繰り返し使うことができます。

4.3.6 感覚

これは個人的意見ですが、「電流１ A は大きいか小さいか」と聞かれたときに安全設計の勘が試されると思います。事実、電子工学分野では 1A はとても大きい電流です。少しでも参考になればと思います。

4.4 モータ制御

私が院試勉強の合間にこっそり考えていたモータ制御に関する安全回路です。

4.4.1 モータのブレーキ

モータって、スイッチを ON にしたら回って、OFF にしたら止まりますよね。でも、その「止まる」というのは本当に止まっているのでしょうか。答えは NO です。スイッチを切ってもモータは惰性で回り続けています。しかもこの間、電磁誘導の法則によって起電力が生まれて回路にダメージを与えてしまいます。そう、安定動作に問題が生じてきます。私はこの点が過去の回路設計における盲点でした。なので今年はこれを修正した回路を考えてみようと考えました。

4.4.2 逆起電力をどう処理するか

逆起電力で生じる電気を GND に逃がす方法を考えました。そこで用いるのが整流作用のあるダイオードです。これは一方向にしか電流を流さないので、逆起電力が発生して逆向きに電流が流れようとした時だけ電流を流すことができます。

4.4.3 ブレーキもつけてしまおう

図**??**を見てください。これはモータとダイオードだけで回路を完結させています。この回路のメリットはモータ自身で電力を消費してくれることに加え、ブレーキが効くということです。つまり、惰性で回転しなくなります。モータに電流を流していて回転状態にしておきます。この状態からスイッチを切るとモータの性質でスイッチを切っても電流を流し続けようとします。しかし、このとき起電力で生じた電流はダイオードを通じることになり起電力は短絡され、一気に 0 に落ち込みます。つまりは惰性の原因となっていた電流が 0 になるの

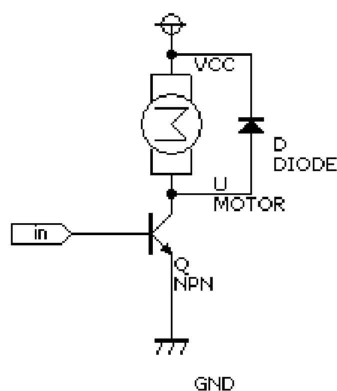


図 4.5: ダイオードを用いた逆起電力対策回路

で回転は消えてしまいます。もうひとつのメリットは、この図ではトランジスタに想定以上の電圧がかからないので素子の保護に役立つということです。回路の安全に一役買っています。

4.4.4 その他の制御回路

他には H ブリッジ回路や、モータドライバ IC というものがあります。原理的には全て同じです。また、モータ以外にも、この考え方はリレーにも応用できます。参考にしてください。

4.5 終わりに

趣味の電子工作ではあまり触れられない回路の安全性について書いてみました。この記事を読んでくださった方には震災以降謳われている安全性について、電子工作にもあてはめて考えてみてほしいと思います。ご存知のとおり私は無類の鉄道好きで電子工作好きです。ここコンピュータ部でも入部してから鉄道関係のテーマばかりを扱ってきました。そのような私ですが来年度からは秋葉原の側に下宿して、大学院にて鉄道をテーマに研究することになりました。いままでは模型でしたが来年度からは本物の鉄道にスケールアップして日々邁進したいと思います。

5　非安定マルチバイブレータで遊ぼう

電子システム工学課程 2 回 松本 駿

5.1　はじめに

最近 LED 光らせることしかやってない気がする松本です。今回はトランジスタの勉強がてら LED を交互点滅させる回路を作りました。

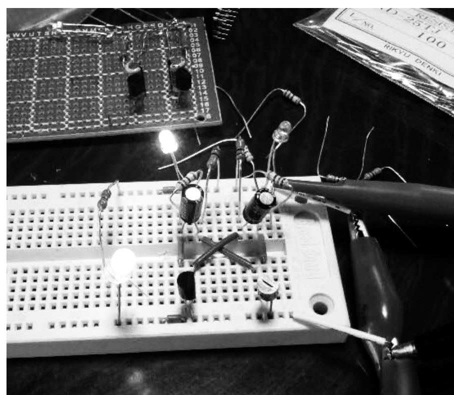


図 5.1: ブレッドボード上での試験

5.2　非安定マルチバイブレータってなに？

非安定 (無安定) マルチバイブレータとは矩形波を出力する発振回路の一つで、抵抗、コンデンサ、トランジスタという簡単な素子を用いて作ることができます。もちろん同じ波形を出力したいだけならマイコンを使ったほうが応用しやすく、半田付けの手間も減ります。

ここではあえて高いパーツを使うのではなく机の上に転がっていたやっすい素子だけでやってみました。使い回しの AC アダプタを除けば使った素子は抵抗 4 つ、可変抵抗 1 つ、コンデンサ 2 つ、2SC1815 を 2 つ、赤色 LED2 つ、基板 1 枚で合計の単価は多分 200 円以下です。ケチにぴったり。
余談ですが Wikipedia[1] によるとマルチバイブレータと呼ばれるのは矩形波に倍音が多く含まれるから、だそうです。

非安定マルチバイブレータ自体は電子工作系列の基本作例でよく扱われるもののようで、回路図などもインターネット上に多数見受けられます。

図 5.2 に一般形の回路図を示しました。言うまでもなく、実際に回路を組む場合は要求に合わせた定数の素子を選ぶ必要があります。また、LED が R1、R4 とそれぞれ直列に配置されていますがこれは単に動作確認のためで回路の構成上必須ではありません。オシロスコープなどで出力される波形を直接確認できるのであれば

[1] Wikipedia-マルチバイブレータ　項より (2011 年 9 月 30 日時点)

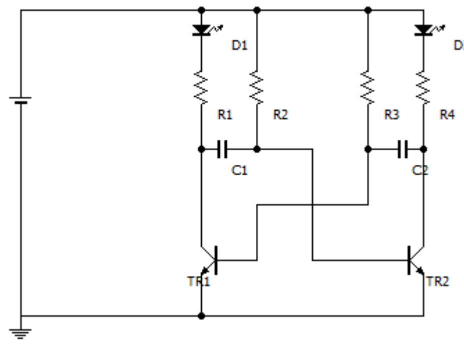


図 5.2: 典型的な非安定マルチバイブレータの回路図

不要です。

5.3 動作原理

さて、なぜ (この回路では LED に) 電流が交互に流れるのか。ここでは n 型トランジスタの、ベース (B) 電流を流すとそれよりもはるかに大きいコレクタ (C) 電流を流すことができるという基本的な性質を利用しています。ベース電流が流れていない時はコレクタ電流も流れません。
電源投入前、コンデンサは放電されていて回路全体は同電位です。

電源投入直後、R1 と R4 により C1、C2 の蓄電が開始され、TR1 と TR2 のベースはどちらも ON になろうとしますが各素子の個体差などの要因でどちらかが先に ON になります。回路は左右対称なのでどちらが先に ON になっても動作自体は変わりません。まず TR1 が先に ON になったとします。

状態 1　TR1 が ON の時

TR1 が ON になると同時に C1 の左端は 0V になります。コンデンサの両端の電圧が一瞬で変動することはないので、左端が 0V に引き下げられると右端も同じだけ電圧が下がり、一気に 0V を通り過ぎてマイナスの値になります。C1 の右端は TR2 のベースに繋がっているので、TR2 の V_B が V_E より低くなるため TR2 は ON になれず、OFF のままです。

状態 2　TR2 が ON の時

一定時間が経過して R2 を介して C1 が蓄電され、TR2 の V_{EB} が 0.6V を超えると TR2 が ON になります。また、C2 の左端がマイナスに引き下げられ、TR1 の V_{EB} が 0.6V 以下になるため TR1 が OFF になります。

この後、C2 が充電され TR1 の V_{EB} が 0.6V 以上になると TR1 が ON になり状態 1 に戻ります。上記の状態 1 と状態 2 の遷移を延々と繰り返すことで、矩形波の出力が得られます。

振動周期は、状態 1 と状態 2 で別々に扱うことができ、R2、R3 の抵抗値と C1、C2 の静電容量値に依存します。基本的には静電容量値が大きければ大きいほど、また抵抗値が大きければ大きいほど蓄電に時間がかかるため状態遷移が遅く、周期が長くなると考えることができます。

状態 1 と 2、それぞれの周期を t_1 と t_2 としたとき、全体の周期 T、周波数 f は以下のように表されます。

$$t_1 = \ln 2 \times R2 \times C1 \qquad \text{(s)} \tag{5.1}$$

$$t_2 = \ln 2 \times R3 \times C2 \qquad \text{(s)} \tag{5.2}$$

$$T = t_1 + t_2 = \ln 2(R2 \times C1 + R3 \times C2) \qquad \text{(s)} \tag{5.3}$$

$$f = \frac{1}{T} = \frac{1}{t_1 + t_2} = \frac{1}{\ln 2(R2 \times C1 + R3 \times C2)} \qquad \text{(Hz)} \tag{5.4}$$

5.4 実構成

5.4.1 定数決定

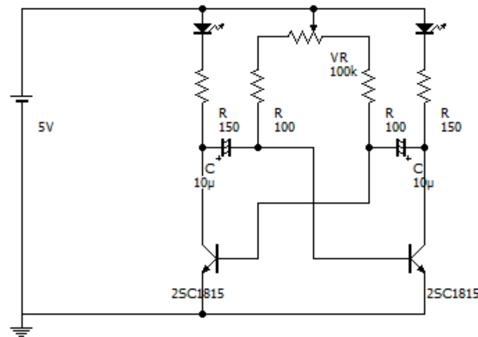


図 5.3: 作成したものの回路図

今回作成したものの回路図は図 5.3 のようになりました。

周期固定だとあんまり面白くないと思い、可変抵抗を使ってデューティ比を可変にしました。全体の周期自体は同一のままで t_1 と t_2 がボリュームの回し具合によって同じ程度増減することになります。

電源は手近にあった 5V の AC アダプタを、R1 と R4 は電源電圧に合わせて赤色 LED が十分光る電流量を確保するため 150Ω を用いています。
n 型のトランジスタは 2SC1815 を選びました。自分が最もよく使うトランジスタで、想定される電流量も電圧も定格内におさまっているのでうってつけです。
デューティ比の変更にあたり、肝心の点滅周期を決めるのは R2 と R3、C1 と C2 なので抵抗か静電容量のどちらかを変化させればいいのですが静電容量可変のコンデンサなんて手元に無いのでここでは 100kΩ 可変抵抗を選択。計算上は、目で見てわかる周波数にするには数十から数百 μF のコンデンサと数十 k から数百 kΩ の抵抗を用いる必要があります。10μF と 100kΩ とだと約 3Hz ほど。

5.4.2 動作確認

まず図 5.1 のとおりブレッドボード上で試作、およそ期待通りの動作を確認できたので可変抵抗以外の素子をユニバーサル基板に半田付けしました。

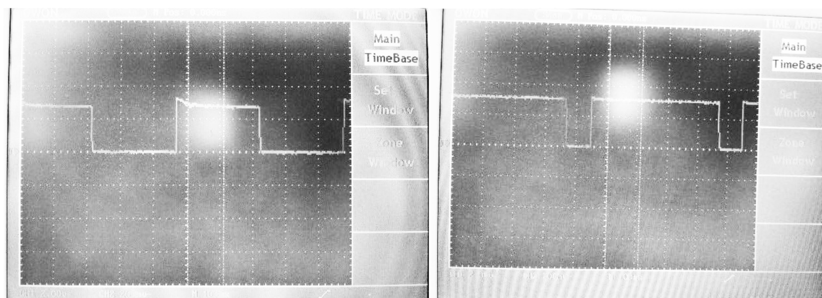


図 5.4: オシロスコープでの波形

D1 の両端にプローブを接続し、オシロスコープで波形を測定しました。図 5.4 の左側が可変抵抗の中心に合わせた時です。ハイ (H・状態 1) とロー (L・状態 2) がほぼ同じ長さになっているのが確認できます。波形の H は 5V、L は 0V(GND) となっています。

次に可変抵抗のボリュームを回して R2 を大きく、R3 を小さくした時の波形が図 5.4 の右側です。H が長く、L が短く、全体の周期自体は同じ長さのままであることがわかります。

計算上はボリュームが中間の時、t_1 と t_2 はどちらも約 0.34 秒、ボリュームをどちらかに振り切った時短い方はほぼ 0 秒 (0.00006s)、長い方は約 0.69 秒となりました。どちらの場合も T は 0.7 秒で周波数 f は 1.4Hz です。図 5.4 のオシロスコープの写真では定数が異なるものを試していたので周波数等はこれと一致しませんが動作としては計算通りの波形を出力できていました。

実際の動作を見ていると、初期状態でチッカッチッカッチッカッといった感じに光っていた LED がボリュームを回すにつれてチッッカチッッカチッッと変化していきなかなか面白い動作となりました。

5.5 終わりに

以上、非安定マルチバイブレータでした。大層な名前の割には簡単です。

素子が少なく、かなりアナログな動作によって実現される回路なので精度が要求される工作においては使えませんが、構成は簡単なのでちょっと点滅させたい時には便利なものです。なによりちょっとだけトランジスタがわかった気になれますし。素子さえあればすぐ製作できるので暇つぶしにでも組んでみてはいかがでしょうか。

参考文献

[1] 趣味の電子工作 目次

http://www.piclist.com/images/www/hobby_elec/

[2] 始める電子工作

http://www9.plala.or.jp/fsson/NewHP_elc/

6 無料でパソコン音楽をやってみたいwindowsユーザーがいわゆる初心者サイトを読む前に目を通してほしい記事。

情報工学課程 2 回 大田 健翔

6.1 はじめに

[1]この記事に目が止まったあなたはおそらく、パソコンで音楽をやることに、少なからず興味のある方なのではないでしょうか。

しかし、パソコンで音楽作りを始めるべく、いざ初心者の為のページへ行ってはみたものの、説明されている中で出てくる「DAW」やら「DTM」やら、プラグインといったなじみのない言葉のオンパレードに恐れおののいてしまう。最初の段階で語彙の壁にぶつかってしまうのです。

例えるなら、英語を学ぶ一番最初の授業から教科書が英語、先生も英語を話して授業をしているような状況、といった感じでしょうか。

DTM を日常で触るような人なら読んで「ああそういうことね」と思うことなのでしょうが、正直 Desk Top Music なんて言われたところで分からない人が大半でしょう。訳して「卓上音楽」。中国語かよというかんじですね。

「どうやったら手っ取り早く音楽が出来るのか分からない。でも、ネットで聞いたら質問厨扱いされて叩かれそうで怖い」。情報モラル、ネチケットのある方ならこう考えるでしょう。そして、よくわからないまま放置→やーめた、となってしまう。

教えて君を淘汰する為に初歩的な質問をする人にキツくあたるのは民度を保持する上で非常に効果的な方法ですが、それと同時に、意欲はあっても調べるノウハウが無いような初学者を疎外してしまうというのも残念なことです。

この記事はそのような「とりあえずパソコンで気軽に音楽をやってみたいけど何をすべきなのか分からない」という方向けの記事です。「始めたい」人向けでないことに注意。それ以前の段階で、右も左も分からない人が対象です。

「イチからはじめる?」だとか「これならわかる?」とか帯に書いておきながら、そこで使われている用語の説明を全くしておらず、文意が理解出来ない悪本を掴まされるという苦い経験をした覚えはありませんか？この記事ではそのようなことが無いように、知識ゼロの方でも、つつがなくパソコンで音楽を作れるように、そして、「やってみる」から「始める」への繋ぎになるように注意を払いながら、軟派雑誌のような口調で説明していきます。

[1]マックユーザーは gallageband というソフトが既にインストールされているのでそれでやる方が早いです。ちなみに Mac ならこの記事のソフトを使ってこの記事通りに音楽を作れます。

6.2 イントロ

パソコンでイラストを描くときのソフトといえば？と言えばSAIやらPhotoshop[2]やらと必ず答えが返ってくるものですが、じゃあパソコンで音楽をやるときのソフトは？と聞くと、たいていの人は「??」というリアクションをするか「えーっと初音ミク？」と答えます。

前者はさしあたってのところ無害なので置いといて、後者は今のうちに認識を改めないと１４，８００円を投資してIYH[3]となったもつかの間、一気にorzとなります。初音ミクは簡単に言えば「ド、レ、ミ…という音程がつけられる機能がついた文章読み上げソフト」であり、「ギターやピアノ、ドラムの音を鳴らして音楽をやる」機能はついていません。

１００歩譲って確かに初音ミクの出す声（音色）でアカペラを作ることを広義的な意味で「音楽をやる」と言えなくも無いのですが、それは、テニスで例えるなら、テニスボールだけ買ってきて手でボールをはたいて遊ぶ行為をテニスだと言っているようなものです。確かにテニスボールを使っている以上、テニスと言えなくもないのですが、傍から見ればうーん…という感じになりますよね。初音ミクは音楽製作のためのソフトとは言いがたいです。

閑話休題。もちろん音楽にも専用のソフトウェアがあるのですが、そういう音楽製作ソフトのことをDAWと言います。次章ではこのDAWって何じゃらほいというのを解説していきたいと思います。

6.3 DAWについて

DAWを略さずに言うと「Degital Audio Workstation」というのですが、正直、知っている人しか意味が伝わりません。

「音楽製作ソフト」と言ってしまうとニュアンスがずれるため、DAWで通っているのだと思います（あくまで憶測）。ちなみに「ダウ」ではなく「ディー・エー・ダブリュ」と言うのが正しい読み方です。[4]そんなDAWですが、今まで通り「音楽製作ソフト」という認識でいいと思います。本来の姿はオーディオファイルを編集する多機能ソフトですが、音楽製作の目的以外でDAWを買う人は多くいないでしょうしね。

さて、上で十分なほどに音楽製作、音楽製作と言ってきましたが、じゃあ、DAWでどうやって音楽を製作していくのでしょうか。次章に移って、実際にDAWを動かしてみましょう。

6.4 無料DAWのダウンロード

[5]今回使うソフトは『Mu.Lab FREE』というDAWです。再三の確認ですが無料です。

ダウンロード方法は検索エンジンで「mu lab」と打てば、よほど下手なことをしない限り「MuTools.com」のHPがヒットするはずです。

リンク先をクリックし、「Downlads」のタブをクリックすれば、『Mu.Lab FREE』のDLページにたどり着くと思います。

[6]後はご自分のOSに合ったバージョンを選んでDLしてください。「Linuxユーザーの俺は…」とか言う人はこの記事のタイトルを読んでいないことがここでバレますm9

ダウンロードできましたか？

インストールはすでに完了していると思うので、ペプシコーラのマークみたいなアイコンをダブルクリックしてみましょう。

[2]後で指摘されたことではあるが、どうやらイラストを描くならPhotoshopよりもIllustratorの方が向いているらしい。

[3]「イヤッッホォォォオオォオウ!!」の略。それなりに大きい金額にもかかわらず欲しくてたまらなくなり、衝動買いしてしまうこと。IYHを叫んだ後、自分の使った金額に絶望しストンと着席してしまうことは世の常。

[4]最近は「ダウ」と言っても伝わるくらいDTMを始める人が増えてきた。しかし初心者のなだれ込む現状を快く思わない古参DTMユーザーの方がとても怒るので「ダウ」と言う際のTPOは注意しよう。

[5]タイトルは「ダウ」と読むと若干韻を踏むことに気づいてない訳ではない。

[6]2011/10/10現在のレイアウトなので、サイトが更新されると、ページデザインに若干の変更がなされるかもしれませんが、ご了承ください。

最初に出てくる CPU が何とかというウインドウは、いじると混乱を招くだけなので、逆らわず素直に OK ボタンを押しときましょうね。

その後に「チュートリアルの載っているサイトを見ますか」という英文ウインドウが出ますが、英語読みながら DAW を弄るのは煩わしいだけですから、逆らわず素直に NO ボタンを押しときましょうね。

さて、いよいよ出てきた DAW という初見キラーなインターフェース。しかし DAW のインターフェースはどれも基本的にこんな感じなので、こいつ１つを押さえておけば、他の DAW をいじるときもさほど苦労はしません。

ってなわけで次章から、実際の「Mu.Lab」の使い方もとい一般的な DAW の扱い方を説明していきます。

6.5　Mu.Lab“で” 学ぶ音楽製作

それにしてもびっくりしますよね、この操作画面。何触っていいのかわかんねーってかすでに曲製作中みたいな感じになってるーっていう印象を持たれた方が大半だと思います。

ああ、ここでやめるって手もアリですよ。前章でぼっちをくらった Linux 難民の相手でもしてやってください。しかし、まずは「やってみる」ってのが大事です。

では、この初期状態から何を触るべきなのかですが、まずはこの既に製作されてるっぽい曲を聞いてみましょう。１番上の再生ボタンをクリックです。（ちなみに、開いたときのサンプル曲はランダムに決まるみたいです。）曲の変更は、左上 FILE OPENSESSION App Library MuSession から好きなやつをどうぞ。

すると曲が再生されます (当たり前)

とりあえず下のなんか荒ぶっているメーターの類いは無視して、上の真ん中についている数字ディスプレイに注目です。

４進法で進んでいるのですが、これは小節のカウントを示しております。

小節というのは、具体的に言えば指揮者のカウントです。合唱コンクールで指揮とっているやつが「１、２、３、４」とか「１、２、３。１、２、３。」というふうにしていますよね。あれが１サイクル終わった時を差します。

１小節中の「１、２、３、４…」の各カウントは１拍。

「１、２、３、４」というふうにカウント出来る曲を４拍子、「１、２、３。１、２、３。」というのを３拍子と言います。

再生している曲では小数点以下で１拍子ずつのカウント、整数値は１小節ずつのカウント、となります。

気づいた方も多いかもしれませんが、この数字、真ん中のごちゃごちゃ書いてあるグラフのなかで右へゆっくりと進む縦長の棒と連動しています。

そして、そのグラフの１番上の行の白地の部分を見てください。１、５、９‥となっており、公差４の等差数列となっておりますが、４小節（４拍じゃないぞ）ごとに区切った群数列ととらえた方がいいかもしれません。（曲によっては公差が違うものもあるようです）

縦棒の表示では４つずつしか小節が記載されていないので、上の数字メーターで再生している位置の詳しい小節を確認する、というのがベターな方法だと思います。

さて、おそらく読者の皆様の中には「何で一般的な mp3 プレーヤーみたいな再生時間が表示じゃねーんだよ」という方もいるでしょう。もっともな意見です。

この１小節ずつの表示だと何がいいのでしょうか。４進法メーターの右肩に注目してください。

なんか１２５とかそこらの数字が書いてあるんじゃないでしょうか。

先に説明してしまうと、これは「BPM(Beat Per Minutes)」と呼ばれる数値です。「１分間に等間隔で何拍打つようなテンポの曲か」を示しています。「いーち、にーい…」という風に１秒ごとに手拍子をすると BPM=60、倍速で「いちにいさんし！」という感じだと BPM=120 となります。分かりにくかったら「数値が大きいほどテンポが速い」と覚えておいてください。

実際にこの右肩のメーターはドラッグで動かせます。どうですか、早くなったり遅くなったりするでしょう？(縦棒の進む早さも変わってますよね)

こういう風にテンポを途中で変更するとき、秒数表示にすると、１秒で２秒分進んでしまったり２秒で１秒（倍速）曲を流してしまったりして、曲の時間を表示するという本来の機能が失われてしまうので、１小節ずつのメーターになっております。

１小節ずつを表示するだけなら「いーち、にーい」でも「いちにいさんし」でも拍は拍で４つで一定。数える速度が違うだけなので、メーターの間隔が変わることはないのです。こういったわけで小節カウンターが採用されています。さて、次章では縦棒の動く画面を説明していきます。

6.6　プレイリストエディター

という風に僕は読んでいますが、DAW（音楽製作ソフトのことですよ。思い出して！）によっては違う名称で呼ばれているかもしれません。

さて、いかにも曲作りの根幹を為していそうなこの部分。どんな機能を持っているのでしょうか。前章に引き続き、サンプル曲を再生してください。

各行の一番左側が楽器の名前になっています。「Piano」「Drum」などと書いてありませんか？

注目して頂きたいのは、曲が色づけされていない部分を、縦棒が通ったとき。

「Piano」列なら着色されてない部分を縦棒が通ったときはピアノの音が消えていませんか？

「Drum」列ならドッタ、ドッタといったリズム楽器の音が消えませんか。

そして、着色部分を縦棒が通ると再び楽器が鳴りだす。

そう、いまあなたが思い描いている認識で正しいです。「何か着色部分が縦棒の上にあるときは、その行の楽器が鳴る」という認識で正しいです。

もうちょい見方を変えるなら「縦棒は色を読み込むスキャナーなんだ」といったところでしょうか。

ではそのスキャナーが読み込んでいる”色”とは何なのでしょうか。次章ではこの色の正体に迫りたいと思います。チャンネルはそのまま！

6.7　ピアノロールと MIDI データとパターンと

この色の部分を見てください。なんだかややツブツブがかってませんか？

ちょっと調べてみましょう。

スキャナーが今にも読み込みそうな色の部分（タイルとでも言った方が分かりやすいかもしれませんね）をダブルクリックしてみてください。

ピアノの鍵盤みたいなやつにちょぼちょぼがつらつらーっとなった画面が現れましたね。

で、スキャナーが通るとそれに合わせて音が鳴っているように思えませんか？

種明かしをしてしまえば、スキャナーは先ほどのプレイリストエディターと一緒なもの。つまり、さっきはタイルを読み込んでいましたが、今度はこのちょぼちょぼを読み込んで音を出しています。

というか、さっきのタイルを拡大したもの（中身という解釈の方が正しい）がこれなんですけどね。気づいた方もいるでしょう。

そして、ちょぼちょぼにも長さが合って、ながいちょぼほど長く音が出ていることに気づくかもしれません。それはスキャナーがそれだけ長く音を読み込んで出していることを考えれば自明の理かもしれませんね。

さて、画面が２分割されていると思います。下の画面は上のちょぼに対応していて、出す音量が変えられます。

つまり、パソコンで曲を作るとは、このように、ピアノの音程に合わせてちょぼちょぼを打ち込んで長さと音の大きさを変えるという行為が基盤になっているのだと考えられます。

さて、ピアノじゃない楽器でもこのピアノみたいな画面でドレミが操れます。このピアノの画面をピアノロールエディターと僕は呼んでいますが、もしかしたら「MIDI ノートエディター」という言い方の方が一般的かもしれません。

このちょぼちょぼのことを MIDI ノートというらしいのですが、正直あまり気にしなくていいと思います。音の音程（ドレミ）、長さ、音量が設定出来る音符だと思えばちょぼちょぼという名前自体さほど問題にはなりません。

ただ、若干興味を持って頂きたいのは、「どの楽器の演奏もちょぼちょぼを打ってスキャナーで読み込めばいい」という一般的な操作ですかね。

こういう、スキャナーで読み込む為のちょぼちょぼを打ち込んで音楽っぽくしたもの、すなわちこのタイルの部分をパターンといいます。

パターンはコピーも出来るので、何枚でも敷き詰めることが出来ます（曲としてはずっと同じパターンを繰り返すので飽きると思いますが)。

曲作りは「ピアノロールでパターンを作る→プレイリストエディターでパターンを並べる」の繰り返しと言ってもいいでしょう。なかなか楽しいんだなこれが。

次章では「Piano」「Drum」のような楽器に焦点を当てていきます。

6.8　ミキサーとモジュールプラグイン

邪魔ですしピアノロール閉じましょうか。ピアノロール左上「Close」クリックです。

さて、したのなんかよくわからない装置の部分の説明をしていきます。

ブロックをクリックするとなんかアクションをするとおもいます。幅が広がるんですね。

その広がった幅を見てみると、何か横長の棒が何本か、つまみが１つ（上にも長方形の何か。)、音楽を再生すると荒ぶるメーターがついていることが分かります。

まず目を向けて頂きたい点は、横長の棒の一番下の段が必ず「MASTER」となっていること、そして、このブロック列の右側に同名の「MASTER」があることですかね。

もちろん偶然ではありません。この「MASTER」が何なのか知っておくことで、音作りの仕組みが分かるようになると思います。

これはミキサーの中のマスタートラックという部分のことです。

はいはい、ミキサーの説明は今からしますからね、あせらないあせらない。

ミキサーと言うからには何かをミックスするのですが、何を混ぜるのかわかりますか？もちろん音に決まってますね（←うざい）

じゃあ、音を混ぜるってどういうことなんでしょうか。

例えば、あるバンドが自分たちが演奏した音楽を録音したいと思ったとき、マイク一本を中央において取ってしまうと、中央に近いボーカルの音声がでかく、ギターやドラムの音は比較的小さくなってしまうことは明らかですね。メンバーからは不満がぷんぷんです。

そこで、かくメンバーの楽器の近くにそれぞれ１本ずつマイクを置いて録音する方法を取ろうと考えました。

さて、録音なのですが、マイクの数だけ録音機を用意しなければいけないのかと言えば当然ノーで、そこで活躍するのが、多数の音を一つにまとめるという役割を持つミキサーの出番なのです。音を拾った各マイクからミキサーへ音が送られ、ミキサーで１つにまとめられます。このミキサーに録音機をつなげれば録音機は１台ですみますね。

上で述べたように、ミキサーは「複数の楽器の演奏した音（信号）を一つにまとめる」機能をもったものです。

では、最初に言った「ミキサーのマスタートラック」とは何かと言えば、複数の音を１つにまとめる部分です。まさしくミキサーのメイン部分ですね。このマスタートラックは録音機へ、もしくはパソコンの音を鳴らす処理をする部分へ音を流します。

さて、「MASTER」はマスタートラックであるということがわかりました。

今、理解してもらいたいことは、「ブロックの１つ１つは楽器がマスタートラックへ音を送っている」ということです。

ついている１つのつまみはマスタートラックへ送る音量

荒ぶるメーターはどれくらいの音量がマスターへ流れているかの目分量

つまみの上の長方形はパンという機能で、楽器の音を左のイヤホン寄りに流すか右のイヤホン寄りに流すか調節する部分です。

どの機能も DAW にはついています。

演奏するために鳴らす楽器のことを DTM(パソコン音楽) ではモジュールまたは音源と言います。モジュールをセットするブロック単位をトラックと言います。

もうちょいエンジニアくさい書き方をすると、各トラックはマスタートラックへ音の信号を送るのです。１トラック１楽器（モジュール）なので覚えておきましょう。ちなみに先ほどあげた初音ミクもモジュールの１種です。ね、初音ミク単体じゃ曲作りが出来ない理屈が分かったでしょ？

実は、DAW にはこういった楽器（モジュール）を追加する機能があります。Mu.Lab の中に入っている楽器じゃ物足りないときにどうぞ、というお話です。

やり方はかんたん。データ拡張子「.dll」の楽器ファイルを App フォルダにぶっ込むだけです。

楽器のデータのことをプラグインと言います。プラグインの追加に関しては、各 DAW によって違いますので、調べる必要があります。が、取扱説明書を理解する知識は既に得ているので心配ないでしょう。

さて、横長に並んだ列に再び注目してください。

何かが間に挟まっているものとそうでないものがあるのが分かりますか？

楽器が音を出して、「MASTER」に送られていく間に何が起こっているのでしょうか。次章ではこいつの正体について解説していきます。が、初めての人が触れると火傷をするところなので、あくまでさらっといきます。

6.9 エフェクトプラグイン

突然ですが y=f(x) の関数の意味は分かりますか？「x を入れると何かの処理があって y という値が生成される」ということですよね。

じゃあちょっと改変して、マスターへ送られる音=f(楽器の鳴らす音) という式は理解出来ますでしょうか。

間に挟まっているのは、この関数 f にあたります。

大まかに分けて関数の種類は以下の７つ。

- フィルター…ある音の高さ（周波数）より大きいものは鳴らさなくする効果。「音を削る」という。例えば、合唱コンクールの曲の高音部分を削ると女子の声が聞こえなくなります。具体的な使い方は、「ジジジジ」という耳障りなサウンドにかけて、「ズズズズ」という若干丸みを帯びた（とがっていない）サウンドに出来ます。が、勘のいい人や、信号処理とかいう分野をかじったことのある人なら、この説明じゃ不十分だと分かるでしょう。vice versa(逆もまた然り)なんですね。つまり、ある音の高さよりも小さいものは鳴らさなくするフィルターも存在します。低音のノイズを除去したいときに使います。前者をローパスフィルター（Low Pass Filter=低い音を・通す・フィルター=高い音は鳴らさないよ）、後者をハイパスフィルター（High Pass Filter=高以下略）と言います。そして、その中間のバンドパスフィルターというやつもあります。高音も低音も削ってしまうので、ラジオや電話機ごしのような音となります。

- コンプレッサー…ある音量よりでかい音を圧縮する装置。その音量より小さい音は圧縮されないので、大きい音と小さい音のバランスがとれる。上のフィルターと違うのは、音の高さ (周波数) ではなくて音の高さ（振幅）である点。女子の声はなくならないが目立とうと大声で歌うやつはコンプで潰されちゃいますよざまぁというお話。

- イコライザ…ある音の高さ（音域）をブーストしたり減らしたりする装置。mp3 プレーヤーについているからわかりやすい。

- ディレイ…やまびこの効果。カラオケでマイクに向かって声を出すと「あっ（あっあっあっ…）」という風にとびとび（離散的）に残響が発生するようなあんな感じ。

- リバーブ…お風呂の効果。つまりは音の余韻が若干伸びる。シンバルの音にこれをかけると、「チッ」と鳴っていたのが「チン」と鳴るようになる。やまびこが連続的かつ高速で返ってくるバージョン（多分この説明じゃわからない）。聞いている空間の広さが調節出来る。

- ディストーション…「楽器の音＋ジャイアン」でだいたいニュアンスが伝わってしまう。ふしぎ！ハードコアやガバといったジャンルの曲では、これをバスドラム（キックという）にかけて、ズンチャカズンチャカやるわけですね。

- フランジャー…名前はかっこいいが、音を左右に揺らすだけ。扱い方を間違えると耳障りになる。

エフェクトも広義的なプラグインなので、楽器と同じく追懐していくことが出来ます。

扱い方が難しい（料理で言うところの調味料）ため、ここでの技術作法は飛ばしたいと思います。

さて、これで曲を作る素養はできました。後は曲を作るだけなのですが、ここからは、既に巷にあふれているネットの情報とかぶる部分になってくるので、そっちを調べれば、少なくとも安っぽい曲はつくることができると思っています。自転車の補助輪を外すときがきたようです。あとは検索エンジンで「初めて　DTM　無料」と検索するだけです。[7]

6.10　終わりに

「

$$f(x,y) + kg(x,y) = 0 \tag{6.1}$$

のグラフは

$$f(x,y) = 0 \tag{6.2}$$

と

$$g(x,y) = 0 \tag{6.3}$$

の交点を通るグラフを表しています。」[8]

高校２年生の時の時の数学を教えていた教師の解説です。

当時僕はこれを聞いて全く分かりませんでした。イメージが全く湧かなかったんですね。

その担任にこれがどういうことなのか聞きにいきましたが「教科書に書いてある通りやろ」と突っぱねられてしまいました。

果たして、上の一文を読んだだけで理解し応用出来る生徒が何人いるのかは分かりませんが、少なくともこの教師の教え方はかなり悪質であることが、予備校の数学を受けていて分かりました。

教科書はあくまで内容を伝える為の手段であり、その教え方は教師に委ねられているため、教師が教科書の言葉をそのまんま引用したところで、生徒が必ずしも理解出来る訳じゃない。

そこで思ったのが「教えるのは簡単だが、内容を分からせるように説明することは難しい」ということです。

そんな経験があるために、どうしても泥臭く回り道をしながら説明してしまうみたいです。

何で人は分からないのかと言えば、基本的な語彙、そしてその単語のイメージが出来ないことが原因だと思っています。

巷にあふれる、DTMの解説サイトも厳密な説明としては正しいのに、用語をそのまま使っているため、その時点で分かりにくいと思ってしまうんですよね。

そういうこともあって、どうやったら知識ゼロの初学者が取扱説明書を読んで理解出来るレベルに持っていけるか。そう考えてこの記事を作りました。

[7]正直、音楽の作り方はパソコンと大きく離れてしまうのでLime的な意味でどうかと思った。
[8]ただ数式を使ってみたくてこのエピソードを出したかったというのもある。

あなたがこの記事を読んで、DTM を独学するにあたっての基盤を築いて頂けたら幸甚なことこの上ありません。

どうもありがとうございました。

参考文献

[1] http://d.hatena.ne.jp/keyword/IYH はてなキーワード「IYH」の項

[2] http://www.mutools.com/ 「MuTools.com」

[3] DTM MAGAZINE vol.193 特集「イチから覚えるエフェクト」 ※表紙の MIKU Append につられて買っても「初音ミク」本体を持っていないと拡張できないので注意する必要が、ってこれ読んでいる人でミク買っている人なんていないですか。失敬。

[4] DTM MAGAZINE vol.195 特集「無料 DTM はじめるガイド」 ※『Lily』の表紙に釣られて買うと見開き１Ｐしか特集されていないことにがっかりするので注意が必要、かも。（参考文献だけみると僕があたかも表紙買いしているような印象を持たれるかもしれませんが違います。会社も商売ですから、純粋な音楽活動家だけでなく、そちらの界隈の人のおサイフも開かせようとしているんですね。）

7 マリオパーティ風ゲーム　マップ編

電子システム 1 回生 判田瑞貴

7.1 企画概要

私は双六のようにマップ上でコマを進めていくゲームを企画しました。コマが止まるたび、そのマスごとに用意してあるイベントが発生します。プレイヤの順序が一周したらプレイヤ全員で戦うミニゲームが行われます。このミニゲームの勝者はゲームを有利に進められるようになります。このゲームはミニゲームと双六の独立性が高いため、チームを組んで複数人で分担して開発しました。私はその内の双六を担当しました。

7.2 メインループ部分の説明

まず SDL_GetTicks() で現在の時間を取得します。これは fps を 60 に合わせるために使用します。次に switch 文で状態遷移を行っております。上から順にタイトル画面、人数選択画面、双六のデータのロード、ゲームプレイ画面、終了処理の状態です。start_menu(),choice_game() にて入力の処理を行います。draw_○○で描画を行います。

```
while(1){
    time_start = SDL_GetTicks();
    switch(main_state){
    case START_MENU:
      start_menu(&start_state,&event_stme,&main_state);
      draw_start_menu(start_state);
      break;
    case CHOICE_GAME:
      choice_game(&main_state);
      draw_choice_game();
      break;
    case LOAD_GAME:
      load_map();
      load_char();
      load_map_common_image();
      main_state = MAP_MAINLOOP;
      break;
    case MAP_MAINLOOP:
      map_mainloop(&main_state);
      break;
    case END:
      SDL_Quit();
```

```
      return -1;
      break;
    }
        省略
  }
```

7.3 ゲームプレイ画面の説明

メニューを開いている状態かどうか判断します。メニューを開けていれば操作をメニューに、開けてなければ操作を双六に移します。次にメニューを開いているかどうかにかかわらず行う描画を行います。そして分岐処理を行い各状態特有の描画を行います。

```
void map_mainloop(){
  if(menu_state == 0){//when you close menu
   maploop_map(引数省略);
  }
  else{//when you open menu
   maploop_menu(&event,&menu_state,main_state,&game_state,&draw_state)
;
  }
  map_draw(x_dif,y_dif,map_move_x,map_move_y);
  char_draw(player_state,game_state,x_dif,y_dif,map_move_x,map_move_y)
;
  menu_back_draw();
  if(menu_state != 0){//when you open menu
    menu_draw(menu_state);
  }
  else if(menu_state == 0 && game_state <= 8){
    map_side_draw(player_state);
    switch(game_state){
    case MAP_WAIT:
      draw_map_wait();
      break;
    case SAI:
      draw_sai();
      break;
    case WAY_CHOICE:
      draw_way_choice(direction,direction_decide);
      break;
    case CHAR_MOVE:
      draw_char_move(direction_decide,player_state,&draw_state,x_dif,
y_dif);
      break;
    case MAP_EVENT:
```

```
      break;
    }
```

7.4 双六のプレイ画面の説明

まずメニューを開くキーを押したかどうか判断します。押した場合は呼び出し側の関数にポインタでそのことを知らせます。次に双六のマップを見渡すモードに切り替えるボタンを押したかどうかを判断します。押した場合はそれに切り替わることを今いるこの関数に知らせます。次に分岐処理を行い現在の状態の処理を行います。上から順に説明しています。基本的に双六の処理は上から順に行われます。

- MAP_WAIT 待機状態です。これは各プレイヤにターンが回ってきたときいきなりサイコロを振る状態になるのは違和感があったのでいれました。
- SAI サイコロを振る状態です。進めるマスが決まります。
- AROUND_SEARCH は進める方向を探します。現在いるマスから四方向に対して進めるかどうかを調べます。
- WAY_CHOICE 進める方向が複数あるとき進む方向をプレイヤに選択させます。
- CHAR_MOVE キャラクタの移動を行います。キャラクタの移動のアニメーション中ならこの処理は行われないようになってます。出た目の数だけ進み終わるまで 3 つ目からこの処理までを繰り返します。進み終われば次は止まったマスのイベントの処理に移ります。
- MAP_EVENT マスのイベントの処理です。事前に登録しておいたイベント用の関数を呼び出します。
- MAP_TARN_END 各プレイヤのターン終了状態です。処理の必要な変数の再初期化を行なってます。
- MAP_GOAL ゴールしたときの処理です。タイトル画面に戻るようになってます。setjump&longjmp を使った理由は状態遷移で処理するより楽だったからです。
- MAP_MOVE 双六のマップ全体を見回せるときの処理です。

```
void maploop_map(引数省略){
  if(event->type == SDL_KEYDOWN && event->key.keysym.sym == SDLK_l
&& *menu_state == 0){
    *menu_state = MAP_MENU_RESTART;//open menu
    event->key.keysym.sym = 0;
    return;
  }
  if(event->type == SDL_KEYDOWN && event->key.keysym.sym == SDLK_m &&
*game_state != MAP_MOVE){
    game_state_store = *game_state;
    *game_state = MAP_MOVE;
    event->key.keysym.sym = 0;
  }
 switch(*game_state){
 case MAP_WAIT://wait time before throw dice
   map_wait(引数省略);
   if(event->key.keysym.sym == SDLK_z){
```

```
      *game_state = SAI;
    }
    break;
  case SAI:// when throw dice
    sai(event,player_state,game_state);
    break;
  case AROUND_SEARCH:
    around_search(event,player_state,game_state,direction_number,
direction,direction_decide,draw_state);
    break;
  case WAY_CHOICE:
    way_choice(event,player_state,game_state,direction_number,direction,
direction_decide,draw_state);
    break;
   case CHAR_MOVE:
    if(*draw_state == CHAR_MOVE)
      break;
    char_move(event,player_state,game_state,direction_number,direction,
direction_decide);
    break;
  case MAP_EVENT:
     mapchip[player[*player_state -1].x][player[*player_state -1].y].
event(event,player_state,game_state,direction_number,direction,
direction_decide);
     *game_state = MAP_TARN_END;
     break;
   case MAP_TARN_END:
    map_tarn_end(event,player_state,game_state,direction_number,
direction,direction_decide);
    break;
  case MAP_GOAL:
    longjmp(buf,1);
  case MAP_MOVE:
    map_move(event,player_state,game_state,direction_number,direction,
direction_decide,map_move_x,m);
    if(event->type == SDL_KEYDOWN && event->key.keysym.sym
== SDLK_m){//when you push "m" again,you get return state.
      *game_state = game_state_store;
      *map_move_x = 0;
      *map_move_y = 0;
      event->key.keysym.sym = 0;
    }
    break;
```

8 ルイージが主役のアクションゲームを作ってみた。

情報工学課程 2 回 葛西 響子

8.1 はじめに

私は、ルイージが大好きで、ルイージが主役となるゲームを作ってみたいと昔から思っていました。前よりもプログラムに関する知識が身についてきたので、念願だったルイージ主役のアクションゲームを作ってみました。

8.2 プログラムの流れ

このゲームがどのように作られたかを抜粋して書いていきたいと思います。このゲームは、DXライブラリ [1] を使って作りました。

8.2.1 ルイージの移動

このゲームは、キーボードの左・右矢印キー (←、→) を押すことでルイージが移動し、上矢印キー (↑) を押すことでルイージがジャンプします。その部分のソースは以下のようになります。

```
void key_input(int *count, int *flag){//自分の移動
  int i;
  int GrHandle;
  static int time1, time2;
  double t;
  if( Key[ KEY_INPUT_RIGHT ]  == 1 ) {//右ボタンが押されたら
    luigi.x+=5;                           //x の値を 5 増やす
    if(luigi.x>=555){
      if( Key[ KEY_INPUT_RIGHT ]  == 1){
        (*count)++;
        luigi.x=0;
        for(i=0;i<3;i++){
          mybullet[i].bullet_x=luigi.x;
          mybullet[i].bullet_y=luigi.y;
        }
      }
    }
  }
  else if(Key[ KEY_INPUT_LEFT ]  == 1){
```

```
      luigi.x-=5;
      if(luigi.x<=0){
        if(Key[ KEY_INPUT_LEFT ]  == 1){
          luigi.x=1;
        }
      }
    }
    if(*flag!=1){
      luigi.y=350;
      GrHandle=LoadGraph("lui.png");
      DrawGraph( luigi.x, luigi.y, GrHandle, FALSE );
    //裏画面へ画像を描写
      ScreenFlip();//裏画面を表画面に反映
    }
    if( Key[ KEY_INPUT_UP ]  == 1 ){//↑が押されたら
      time1 = GetNowCount();//time1 に↑が押された時の時間を格納
      *flag=1;              //飛び上がりフラグを立てる。
    }
    if(*flag==1){
      time2 = GetNowCount();        // 現在経過時間を得る
      t = (double)(time2 - time1) / 1000.000;
   //ミリ秒を秒に変換して、↑が押されてからの経過時間を計算
      luigi.y = (int)((sqrt ( 2.000 * G * Y_MAX) *
                t - 0.500 * G * t * t ) * 286.000 /
                Y_MAX);//y 座標を計算
      if(luigi.y>=0){//1 回目に回って来たか、画面内にｙ座標がある時
        GrHandle=LoadGraph("lui.png");
        DrawGraph( luigi.x, 350-luigi.y, GrHandle, FALSE );
        //画像を描画
        ScreenFlip();//裏画面を表画面に反映
      }
      else
        *flag=0;//画面外に来ると、飛び上がりフラグを戻す
    }
  }
```

ルイージが右に移動して画面外に行きそうになったら、ルイージの位置が画面の左端に移動します。あたかもルイージが別の画面に移動したように見せています。ルイージが左に移動して画面外に行きそうになったら、ルイージの位置を画面の左端に固定し、左矢印キーを押してもそれ以上左に移動できないようにしています。ルイージがジャンプしているように見えるように、フラグを使って、表示するルイージの画像の位置を変えています。ルイージの y 座標を計算する時には、物理の公式 (等加速度運動の公式) を用いています。

8.2.2 ルイージのライフと表情の表示

ルイージのライフと表情の表示について説明します。この部分のソースは、以下のようになります。

```
int show_life(void){//ライフの切り替え
```

```
  if(luigi.life==3){
    DrawCircle( 20, 20, 10, GetColor(255, 0, 0), TRUE );
    DrawCircle( 60, 20, 10, GetColor(255, 0, 0), TRUE );
    DrawCircle( 100, 20, 10, GetColor(255, 0, 0), TRUE );
    LoadGraphScreen( 500, 0, "lui_life3.png", FALSE );
  }
  else if(luigi.life==2){
    DrawCircle( 20, 20, 10, GetColor(255, 0, 0), TRUE );
    DrawCircle( 60, 20, 10, GetColor(255, 0, 0), TRUE );
    DrawCircle( 100, 20, 10, GetColor(0, 0, 0), TRUE );
    LoadGraphScreen( 500, 0, "lui_life2.png", FALSE );
  }
〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜省略〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜
  else{
    DrawCircle( 20, 20, 10, GetColor(0, 0, 0), TRUE );
    DrawCircle( 60, 20, 10, GetColor(0, 0, 0), TRUE );
    DrawCircle( 100, 20, 10, GetColor(0, 0, 0), TRUE );
    LoadGraphScreen( 500, 0, "lui_life0.png", FALSE );
    DrawString(300, 30, "ゲームオーバー",
               GetColor(255, 255, 0));
    return 1;
  }
  return 0;
}
```

ルイージのライフがいくつであるかを判定し、その値によってライフの数やルイージの表情を変えます。DrawCircle 関数で円を描画できるので、それを使って残りライフを表示します。LoadGraphScreen 関数でルイージの表情が描かれた画像を読み込めるので、それを使ってルイージの表情を表示します。

8.2.3 ルイージの弾の発射

ルイージがどのように弾を発射しているのかについて説明します。
この部分のソースを以下に示します。

```
void bullet(int *counter, int flag){//自分の弾の移動
  int i;
  if(*counter<5){
    (*counter)++;
  }
  else if( Key[ KEY_INPUT_RETURN ]  == 1 ){
    *counter=0;
    for(i=0;i<3;i++){
      if(mybullet[i].bullet_flag==0){//発射していない玉を探し、
        mybullet[i].bullet_flag=1;   //発射フラグを立てる
        mybullet[i].bullet_x=luigi.x;
        if(flag==1){
          mybullet[i].bullet_y=350-luigi.y;
```

```
        }
        else{
          mybullet[i].bullet_y=luigi.y;
        }
        break;
      }
    }
  }
  for(i=0;i<3;i++){
    if(mybullet[i].bullet_flag==1){//発射している玉なら
      mybullet[i].bullet_x+=8;      //座標を 8 増やす
      DrawCircle( mybullet[i].bullet_x+30,
                  mybullet[i].bullet_y+30,
                  10, GetColor(0, 255, 0), TRUE );//玉を描画
      if(mybullet[i].bullet_x >= 680){//もし画面外まで来たら
        mybullet[i].bullet_x=luigi.x;
        mybullet[i].bullet_y=luigi.y;//初期値に戻し、
        mybullet[i].bullet_flag=0;    //発射フラグを戻す
      }
    }
  }
}
```

フラグを用いて、ルイージが連続で弾を 3 発まで打てるようにしています。弾の位置座標を増やし、DrawCircle 関数で弾を描画することで弾が飛んでいるように見せています。弾が画面外まで移動したら、もう一度ルイージが弾を打てるようになります。また、ルイージがジャンプしている時でも、その位置から弾が打てるようにしています。

8.2.4　ルイージと敵の当たり判定

ルイージが敵と接触したかどうかを判定します。この部分のソースは、以下のようになります。

```
void atari_hantei_luigi_enemy(int count){//当たり判定
  int i;
  int hit_x[10], hit_y[10],
      hit_range=luigi.range+enemy[0].range,
      hitb_range=luigi.range+boss.range;
  if(count==0){
    hit_x[0]=luigi.x-enemy[0].x;
    hit_y[0]=luigi.y-enemy[0].y;
  }
  else if(count==1){
    for(i=1;i<=2;i++){
      hit_x[i]=luigi.x-enemy[i].x;
      hit_y[i]=luigi.y-enemy[i].y;
    }
  }
```

```
〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜省略〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜
    else if(count==4){
      hit_x[6]=(luigi.x+30)-(boss.x+100);
      hit_y[6]=(luigi.y+30)-(boss.y+100);
    }
    if(count==0){
      if(enemy[0].flag!=0&&hit_x[0]*hit_x[0]+hit_y[0]*hit_y[0]
                                         <=hit_range*hit_range){
        if(luigi.x<=enemy[0].x){
          luigi.x-=50;
        }
        else{
          luigi.x+=50;
        }
        luigi.life--;
      }
    }
    if(count==1){
      for(i=1;i<=2;i++){
        if(enemy[i].flag!=0&&
           hit_x[i]*hit_x[i]+hit_y[i]*hit_y[i]
                                         <=hit_range*hit_range){
          if(luigi.x<=enemy[i].x){
            luigi.x-=50;
          }
          else{
            luigi.x+=50;
          }
          luigi.life--;
        }
      }
    }
〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜省略〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜
    if(count==4){
      if(boss.flag!=0&&hit_x[6]*hit_x[6]+hit_y[6]*hit_y[6]
                                       <=hitb_range*hitb_range){
        luigi.x-=100;
        luigi.life--;
      }
    }
  }
```

三平方の定理より、ルイージと敵の x 座標の差の 2 乗と、ルイージと敵の y 座標の差の 2 乗を足したものがルイージと敵の半径の和の 2 乗以下であれば、ルイージが敵に接触したといえます。ルイージが敵と接触していた場合は、ルイージが少し吹き飛ばされ、ルイージのライフが減ります。ルイージの弾と敵の当たり判定や、敵の弾とルイージの当たり判定、ルイージの弾と敵の弾の当たり判定も同様の原理を用いています。

8.3 ゲーム動作画面

このようなゲーム画面になります。



8.4 今後の課題

キャラクターの透明化がうまくいっていないところや、当たり判定がずれているところ、ルイージの移動が遅いところを直していきたいと思います。また、コントローラーで操作できるようにしたいと思います。

8.5 おわりに

ゲームのプログラミングは、やはり難しいことが改めてわかりました。でも、念願のルイージ主役のアクションゲームが作れて、達成感でいっぱいです。今後も頑張って勉強して、このゲームをパワーアップさせたり、もう一歩進んだゲームを作ったりしたいと思います。

参考文献

[1] Remical Soft
Ｃ言語〜ゲームプログラミングの館〜[ＤＸライブラリ]
< http://dixq.net/g/index.html >

9 SDL を使った創作ミニゲーム

情報工学課程 1 回 中川 鴻佑

9.1 制作の動機

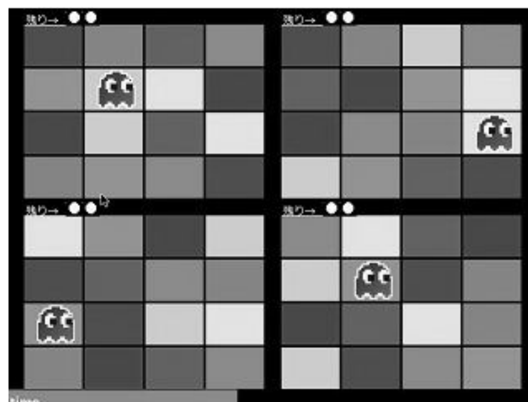
前期の終わりに 1 回生でチームを組んで 1 つのゲームを作ろうという企画が持ち上がりました。私はコンピュータ部に入部する前からゲーム制作に興味を持っていたので、SDL の勉強もかねて企画に参加しました。

そのゲームはマリオパーティのような、すごろくとミニゲームを組み合わせたパーティゲームで、私はその中のミニゲームの担当になりました。

9.2 ミニゲームの概要

ランダムに色が割り振られたパネルの上を上下左右に動きながら、自分が居るパネルと同じ色のパネルにいる他のプレイヤーを落としていくゲームです。

プレイヤーの数は 4 人で、各プレイヤーには 4 × 4 = 16 マスのパネルが割り当てられ、その中を移動できます。パネルは全 8 色で、どの色のパネルも数に偏りがありません。また、一人のプレイヤーが移動できる範囲の中には、同じ色のパネルが 2 つずつあり、ランダムに配置されます。



プレイヤーは移動の他に 1 つだけアクションを起こすことができます。アクションを起こすと、プレイヤーの足元のパネルと同じ色のパネルがすべて開き、他のプレイヤーがその上にいた場合、そのプレイヤーは落下します。3 回落下したプレイヤーは失格となり、より長く生き残れたプレイヤーの勝ちとなります。

9.3 SDL とは

SDL(Simple Direct Media Layer) とはグラフィックやサウンドなどを使ったソフトウェアを簡単に開発することができるようにしてくれるライブラリです。今回はウィンドウの表示とグラフィックの表示、そしてキー入力の受取に SDL を使いました。

また、SDL を使う際には SDL.h をインクルードします。またはグラフィックを扱う際には SDL_image.h をインクルードします。

9.4 プレイヤーおよびパネルに関する構造体

ここからは SDL についての解説を入れながら、処理ごとにソースコードを抜粋して紹介していきます。
まず構造体についてです。プレイヤーに関する情報を格納する構造体 Player と、パネルに関する情報を格納する構造体 Panel を作りました。データの中身については必要に応じて後々説明します。

```
typedef struct Factor1{
    int x;
    int y;
    SDL_Surface *img;
    int prg;
    int count;
    int hp;
    int rank;
    int inv;
    int inv_t;
}Player;
typedef struct Factor2{
    int x;
    int y;
    SDL_Surface *img;
    int c_num;
    int t_c_num;
    int prg;
    int count;
}Panel;
Player player[4];
Panel panel[4][4][4];
```

配列の player は 4 人のプレイヤーのデータを格納し、配列 panel はプレイヤー 4 人×パネル 4 列×パネル 4 行＝64 枚のパネルのデータを格納します。

9.5 ランダムにパネルを配置するプログラム

まず、mix という関数を用意し、配列 r[8] に 0〜7 の 8 つの数をランダムに重複なく格納します。

```
void mix(int *r){
    int j,x,y,tmp;
    for(j=0;j<8;j++)
        r[j]=j;
    for(j=0;j<1000;j++){
        x=rand()%8;
        y=rand()%8;
        tmp=r[x];
```

```
        r[x]=r[y];
        r[y]=tmp;
    }
    return;
}
```

1 つ目の for ループでは r[8] に 0 から順に数字を格納します。2 つ目の for ループですが、rand は出鱈目に自然数を返す関数なので、rand()%8 は 0〜7 のいずれかの数を返します。このループでは r[0]〜r[7] からランダムに 2 つを選んで、格納されている数字を入れ替えるという処理を 1000 回行います。こうすることで、r[0]〜r[7] に重複しない 0〜7 の数がランダムに得られます。

次に panel_init 関数の処理によってパネルをランダムに配置します。

```
void panel_init(){
    int i,j,k,l,r[8];
    srand((unsigned)time(NULL));
    for(i=0;i<4;i++){
        mix(r);
        l=0;
        for(j=0;j<4;j++){
            for(k=0;k<4;k++){
                panel[i][j][k].img=color[r[l]];
                panel[i][j][k].c_num=r[l];
                panel[i][j][k].t_c_num=r[l];
                l++;
            }
            if(j%2==1){
                l=0;
                mix(r);
            }
        }
    }
    return;
}
```

mix 関数では擬似乱数を返す rand という関数を使いました。rand は種と呼ばれる初期値をもとに計算を行い、出鱈目な数を作ります。種は srand で指定します。種として現在の時刻を指定することにより、rand はプログラムを実行する度に違う数を返します。

次にこの関数で使われる変数についてです。配列 color は SDL_Surface 型のポインタ配列であり、SDL_Surface 型とは画像を格納するデータ型です。color という名の通り、color[0]〜color[7] はパネルの色 (全 8 色) を表しますが、color[0]〜color[7] はポインタなので 8 色のパネルの画像のアドレスが格納されています。(ポインタを使うのは画像を表示する際に便利だからです) また構造体変数 panel のメンバ c_num と t_c_num には color[ ] の [ ] 内の数、即ちパネルが何色かを示す番号 (これを色番号とする) が格納されます。この番号は後で使用します。

最後に具体的な処理として、先ほど紹介した関数 mix で配列 r に乱数を格納し、panel[ ][ ][ ].img に color[r[ ]] を代入します。配列 r には乱数が格納されているので、color[r[ ]] はランダムな色を表しており、したがってこの処理は「ランダムにパネルを配置する」ことにあたります。

9.6 プレイヤーのアクションによりパネルを開くプログラム

プレイヤーがアクションを起こした後の処理です。panel_reverse という関数で動作させます。

```
void screen_point_to_panel_point(int x, int y, int player_no, int *px, int *py) {
    int x_masu, y_masu;
    x -= (player_no % 2) * 290;
    if(player_no > 1) {
        y-=210;
    }
    x_masu = x / 73;
    y_masu = y / 53;
    *px = x_masu;
    *py = y_masu;
}

int screen_point_to_color(int x, int y, int player_no) {
    int x_masu, y_masu;
    screen_point_to_panel_point(x, y, player_no, &x_masu, &y_masu);
    return panel[player_no][y_masu][x_masu].c_num;
}

/*パネルを開く*/
void panel_reverse(int i){
    int j,k,l,change;
    if(player[i].x_move==0&&player[i].y_move==0){
        change = screen_point_to_color(player[i].x, player[i].y, i);
        for(j=0;j<4;j++){
            for(k=0;k<4;k++){
                for(l=0;l<4;l++){
                    if(panel[j][k][l].c_num==change){
                        panel[j][k][l].img=pacolor[change];
                        panel[j][k][l].c_num=i+8;
                    }
                }
            }
        }
        player[i].prg=1;
    }
    return;
}
```

panel_reverse は引数に int 型の数をとりますがこれは player[i] の i の値です。つまりこの関数での処理はプレイヤー一人分の処理ということになります。

screen_point_to_panel_point はプレイヤーの位置を特定する関数です。ここでプレイヤーの位置を示す player[i].x・player[i].y ですが、このメンバの値はゲーム画面の左上に原点、右方向に x 軸、下方向に y 軸をとった時の座標です (単位はピクセル)。player[i].x-(i%2)*290 や y-=210 という処理は画面左上からの距離で表される座標を各プレイヤーの移動範囲の左上からの距離に直すためのものです。そうして得られた値 x、y をそれぞ

れパネルの横の長さ・縦の長さで割って、プレイヤーが左上から何マス目のパネルにいるかを調べます。

screen_point_to_color はプレイヤーが何色のパネルの上にいるかを調べます。具体的にはプレイヤーが居るパネルの色番号を返します。

panel_reverse はこれらの関数を使って、パネルの色を調べ、プレイヤーがアクションを起こしたパネルと同じならばそのパネルを開きます。具体的には各パネルの色番号と変数 change とを比較し、同じであれば構造体メンバ img に別の画像のアドレスを入れ、またパネルの色番号をプレイヤーに対応したものに変えます。「別の画像のアドレス」として、ここでは p_color[i] を使用しています。色番号をプレイヤーに対応したものにしているのは、アクションを起こしたプレイヤー自身が落下しないようにするためです。また別の関数で、開いたパネルが一定時間経過後に元に戻るようにしています。

最後にプレイヤーが一度アクションを起こしたら、一定時間アクションを起こせないようにします。player[i].prg に 1 を格納する処理です。main 関数では player[i].prg が 1 未満でなければアクションを起こせないようになっています。また別の関数で、一定時間経てば player[i].prg が 0 に戻るようにもしてあります。プレイヤーがアクションを起こせない時間は開いたパネルが元に戻るまでの時間よりも長くしているので、プレイヤーがパネルを開きっぱなしにすることはできず、無敵になることはありません。

9.7 プレイヤーが落下し、残機がなくなったプレイヤーから順位を決定するプログラム

player_fall という関数でプレイヤーが落下し、残機がなくなったプレイヤーから順位を決定するという動作を行います。最初に screen_point_to_panel_point を使って、プレイヤーの位置を特定します。また、この関数にはプレイヤーが残り一人になるとゲームを終了する処理も入っていますが、簡単な処理なので割愛し、処理の中心となるところだけを紹介します。

```
for(i=0;i<4;i++){
    screen_point_to_panel_point(player[i].x, player[i].y, i, &x_masu, &y_masu);
    if(panel[i][y_masu][x_masu].c_num>7&&panel[i][y_masu][x_masu].c_num!=i+8)
        if(player[i].inv==0){
            player[i].hp-=1;
            player[i].inv=1;
        }
        if(player[i].hp==0){
            player[i].img=NULL;
            player[i].rank=rank;
            player[i].hp-=1;
        }
    }
}
```

まず、プレイヤーが居るパネルの色を調べます。ソースの一番長い行です。アクションを起こしたプレイヤーの構造体が player[n] のとき開いているパネルの色番号は 8+n(前章参照) だから、色番号が 8 未満か 8 以上かを調べることで、パネルが開いているかどうかがわかります。そして色番号が自分に対応したもの (ここでは i+8) でなければ、プレイヤーが落下します。

プレイヤーが落下するとプレイヤーの残機が減ります。if(player[i].inv…で始まる if 文を見てください。残機数は player[i].hp に格納されているので、player[i].hp から 1 引きます。また、落下した後に一定時間の無敵時間を設けます。player[i].inv が 1 になっているうちは player[i].hp が減りません。ただし player[i].inv は別の関数の処理により一定時間経つと 0 になります。

そしてプレイヤーの残機が 0 になるとプレイヤーは失格になり、プレイヤーのグラフィックも消えて、いかなる操作もできなくなります。そして順位が決定されます。if(player[i].hp…で始まる if 文を見てください。player[i].img を空にし、player[i].rank に、次に失格になるプレイヤーの順位が格納されている変数 rank を代入します。そしてこの処理が繰り返されないように player[i].hp からまた 1 を引きます。

9.8 メインループ

ここまでは自作関数の説明をしてきましたが、ここからは main 関数の中でもゲームのメインループを大きいループから解説します。

```
SDL_Event event;
exit_prg=1;
while(exit_prg){
/*終了条件が満たされるまでループを回す*/
    if(SDL_PollEvent(&event)){
    /*処理されてないイベントを処理する*/
            switch(event.key.keysym.sym){
            /*キーに応じて処理*/
                default:
                    break;
            }
    }
    /*プレイヤーがそれぞれの移動範囲から*/
    /*はみ出ないようにする。*/
    player_seal();
    /*プレイヤーが落ちる*/
    player_fall();
    /*描画*/
    draw();
    /*ちょっと待って裏返ったパネルを戻す*/
    panel_return();
    SDL_Delay(10);

}
```

まず、ゲームの終了条件を満たすまでループを回し続けるために、変数 exit_prg を用意します。exit_prg には 1 を格納しておき、処理の中で終了条件が満たされると 0 になるようにしておきます。

次にすべての入力に対して適切に処理するために、if(SDL_PollEvent(&event)) を使います。SDL_Event 型の構造体変数 event はキー入力をイベントとして受け取る型の変数で、関数 SDL_PollEvent は処理されていないイベントがあれば 1 を返す関数です。これにより同時に複数のキーが押されても処理が飛ばされることなく、メインループが回る度に順番に処理されていきます。

そしてどのキーが押されたかを判別し、キーに応じた処理を行います。構造体メンバの event.key.keysym.sym にはキーシンボルと呼ばれるものが格納されます。例えば a キーを押したときには SDLK_a、ENTER キーを押したときには SDLK_ENTER という具合です。

キー入力による処理が終わったら後始末です。解説したもの以外にもいくつか自作関数を使っています。

まず、キー入力によってプレイヤーが移動範囲から出た場合にプレイヤーを引き戻す処理を player_seal で行います。

次にプレイヤーがアクションを起こしてパネルが開いていたら player_fall でプレイヤーを落とします。

この時点でプレイヤーの残機があるかどうかや、プレイヤーの位置が決定しているので、パネルとプレイヤーの画像を draw でスクリーンに描画します。

そして、開いたパネルを元に戻す、アクションを起こしたプレイヤーが連続でアクションを起こせないようにする、落下したプレイヤーが無敵になるなど、一定時間のみ有効な処理を panel_return で行います。

最後に 100 分の 1 秒待機するという処理を入れておきます。SDL_Delay は引数の値の分だけミリ秒単位で待機する関数です。この処理がなければ、CPU がフル回転することになってしまいます。

続いて上のコードの/*キーに応じて処理*/の中身を見ていきます。

まずはプレイヤーが移動する処理です。例えば a キーを入力によってプレイヤーが左に移動する場合は以下のようになります。

```
case SDLK_a://a キーで
    if(player[0].hp>0)
        player[0].x-=PANEL_WIDTH;//1P 左へ
    break;
```

PANEL_WIDTH はパネルの横幅を表します。右に移動する場合は player[0].x+=PANEL_WIDTH となり、上に移動する場合は player[0].y-=PANEL_HEIGHT となります。また、プレイヤーの残機が 0 のときは処理を行いません。

次にパネルを開く処理です。x キーの入力によってプレイヤーがパネルを開く場合は以下のようになります。

```
case SDLK_x: //x キーで
    if(player[0].hp>0&&player[0].prg<1)
        panel_reverse(0);
    break;
```

プレイヤーの残機が残っていて、アクションを起こしてから一定時間が経っていれば、パネルを開きます。

9.9　感想

今回の制作で、練習としてやっていたプログラミングとは違った、より実用的なプログラミングについて学ぶことができました。SDL の使い方はもちろん、以前はあまり使っていなかった構造体や自作関数についても身につけられたと思います。

ただしまだゲームとしての完成度は低いのでこれから改良を加えていきたいです。

編集後記

Lime44 号、お楽しみいただけたでしょうか。編集作業という仕事をするのは人生で初めての経験だったため、うまくできるのかどうか不安でしたが、作業に知恵をかしてくださった部員達の力で、なんとか形にすることができました。この場をかりてお礼を申し上げます。TEX に慣れていなかったため、いろいろ苦労しましたが、編集作業を通して TEX に対する理解が深まり、いい機会になったと思います。
ここまで読んでいただき、ありがとうございました。

平成 23 年 11 月 20 日 編集担当 葛西 響子